Manuale pratico di Java

**La teoria integrata dalla pratica:
dalle basi del linguaggio alla programmazione distribuita,
un percorso collaudato per apprendere la tecnologia Java**

P. Aiello L. Bettini L. Dozio A. Gini A. Giovannini M. Manzini
M. Molino G. Morello G. Puliti S. Rossini N. Venditti

Manuale pratico di Java
La teoria integrata dalla pratica: dalle basi del linguaggio alla programmazione distribuita, un percorso collaudato per apprendere la tecnologia Java

I materiali di approfondimento e gli esempi cui si fa riferimento nel testo sono disponibili all'indirizzo http://www.mokabyte.it/mokabook

via Baracca, 132
50127 Firenze
http://www.mokabyte.it
e-mail: info@mokabyte.it

viale Montenero, 44
20135 Milano
http://www.hopslibri.com
e-mail: hops@hopslibri.com

ISBN 88-8378-039-6

Immagine di copertina: grafica Hops Libri da un'idea di Lorenzo Pacini
Redazione e impaginazione: Francesco Saliola su progetto tipografico Hops Libri

Hops Libri è un'iniziativa di Metafora srl

Indice generale

Prefazione

Nel presentare brevemente ai lettori questo *Manuale pratico di Java* è necessario anzitutto ripercorrere il cammino che ha portato alla ideazione e alla realizzazione dell'opera.

In primo luogo va rimarcato che si tratta di un libro scritto da più autori, coinvolti a vario titolo nelle attività di MokaByte (`http://www.mokabyte.it`), rivista web in lingua italiana dedicata al mondo Java. Ciò ha consentito di presentare i molteplici argomenti in maniera approfondita grazie alle competenze specifiche e alle esperienze dirette di ciascun autore.

Annate di pubblicazione della rivista web, l'esperienza maturata in corsi e consulenze a vari livelli, il rapporto diretto con lettori e allievi hanno contribuito a far sì che si accumulasse una notevole mole di materiale didattico. Occorreva uno sforzo di selezione, organizzazione, aggiornamento e revisione affinché da tali risorse potesse nascere un "manuale" in grado di permettere un facile apprendimento dei molti argomenti legati a Java: l'opera è stata pertanto realizzata integrando la teoria con spiegazioni pratiche ed esempi concreti.

L'organizzazione modulare dei vari capitoli fornisce un percorso collaudato per passare dagli argomenti più semplici a quelli più avanzati, ma è adatta anche a chi desideri approfondire solamente alcuni temi specifici. Il taglio eminentemente pratico del libro, comunque, non perde mai di vista la vera filosofia alla base della programmazione Java.

Ci auguriamo che, nel panorama sempre più agguerrito e, in qualche caso, "nevrotico" dell'editoria per l'informatica, l'impostazione e i contenuti di *Manuale pratico di Java* possano rappresentare un utile contributo a una maggiore conoscenza e diffusione delle tecnologie Java.

Un doveroso ringraziamento va rivolto alla società Imola Informatica per aver creduto nel progetto e per averlo sostenuto con attenzione e competenza durante la lunga fase di gestazione.

Al termine della sua realizzazione, *Manuale pratico di Java* va in stampa con dimensioni quasi doppie rispetto a quanto inizialmente preventivato: ciò ha creato imprevisti e ritardi che l'editore ha accettato comunque con comprensione; per la pazienza e la simpatia, e non solo per la professionalità e il dinamismo, desideriamo ringraziare tutti i componenti dello staff di Hops Libri, la cui collaborazione è stata per noi piacevole e proficua.

La pratica ci insegna che i libri sono entità "vive" le quali si nutrono delle critiche e delle correzioni, delle precisazioni e dei consigli che giungono dai lettori. Saremo grati a quanti vorranno segnalare mancanze e refusi, nell'ottica di un miglioramento dell'opera.

la Redazione di MokaByte

Introduzione a Java

di Giovanni Puliti

Java: il linguaggio e la tecnologia

Chi si avvicina a Java per la prima volta ne resta al contempo affascinato ma a volte anche piuttosto confuso, data la vastità degli argomenti che compongono il panorama legato a questa tecnologia.

In questo capitolo verrà fatta una breve panoramica sui concetti introduttivi legati a Java, in modo da poter iniziare a scrivere i primi programmi: al termine dovrebbero essere stati acquisiti gli elementi basilari di pratica e teoria tali da permettere di addentrarsi nella teoria vera e propria esposta nei capitoli successivi.

Breve storia di Java

La linearità ed il rigore nell'organizzazione di Java nascono paradossalmente da un percorso lungo e piuttosto travagliato che ha presentato cambiamenti di direzione, continue ridefinizioni degli obiettivi, momenti di stasi e brusche accelerazioni, grandi entusiasmi sostenuti da un lavoro spesso frenetico e spossante.

Il tutto ebbe inizio nel 1988 quando in Sun venne fatto partire un progetto per un nuovo sistema di interfaccia grafica, denominato NeWS (*N*etworked/*e*xtensible *W*indows *S*ystem), basato su un concetto nuovo per l'epoca: maggiore astrazione rispetto all'hardware sottostante, sia in termini di riferimenti alle risorse, sia dal punto di vista logico. La scelta iniziale del PostScript per la visualizzazione delle finestre aveva proprio questo scopo. Protagonisti di questo progetto furono James Gosling e Patrick Naughton, sostenuti da Bill Joy.

I primi due anni furono spesi nell'impresa di integrare il sistema NeWS di Sun Microsystem con X11 nel tentativo di raggiungere una maggiore portabilità: tale sforzo si rivelò impossibile.

A prescindere dai risultati del lavoro, questa esperienza fece maturare nella testa dei personaggi sopra citati l'importanza della necessità di una soluzione nuova al problema dell'astrazione dall'hardware sottostante e della portabilità.

Abbandonato il progetto NeWS, in Sun si decise verso il 1990 di far partire un progetto gestito indipendentemente da un piccolo gruppo esterno. La filosofia principale era la massima autonomia dalle scelte politiche e tecnologiche di Sun. Non ci si sarebbe appoggiati a nessuna piattaforma in particolare ed anzi, dopo aver ipotizzato l'utilizzazione della piattaforma PC, si pensò alla realizzazione di un sistema fortemente orientato alla grafica ed alla multimedialità, basato su hardware proprietario,

Il progetto si sarebbe rivolto al mercato di largo consumo, avrebbe dovuto avere una interfaccia grafica attraente ed al tempo stesso innovativa. La prima mossa fu quella di realizzare un computer portatile basato su tecnologia simile a quella di Sun.

Il kernel del sistema operativo era un derivato del SunOS riadattato in modo che potesse girare su un sistema molto più piccolo: questo fu il progetto Green, predecessore di Java.

Nei 18 mesi del suo svolgimento, il gruppo di programmatori dividerà le proprie giornate fra lunghe ore di lavoro, videogiochi, partite serali a hockey durante le quali spesso si discuteva di lavoro e si traeva spunto per le idee innovative da implementare di giorno — discussioni e litigi che porteranno a diverse modifiche nella struttura del gruppo.

Alla fine dell'estate 1992 vene presentato il prototipo hardware Star7, una specie di supporto per applicazioni di largo consumo, come cercapersone o videoregistratori programmabili. Basato su un una versione alleggerita del SunOS, il progetto dette vita contemporaneamente ad Oak (il padre di Java) un linguaggio a oggetti con una spiccata capacità per la gestione della multimedialità e delle comunicazioni; il livello di astrazione rispetto all'hardware sottostante con cui Oak era stato realizzato era un concetto sicuramente nuovo per l'epoca.

Il prototipo hardware realizzato doveva essere un surrogato di un elettrodomestico a diffusione di massa: nei 4 megabyte di memoria RAM erano installati il SunOS, l'interprete Oak, la libreria grafica, le classi dell'interfaccia utente, le applicazioni, le immagini, le animazioni e i suoni. Gli obiettivi del progetto Green erano stati raggiunti ma ciò non implicava necessariamente la diffusione massiccia di tale tecnologia su larga scala.

Purtroppo un mercato non pronto a digerire la diffusione di una macchina tecnologicamente così avanzata, unitamente alla concorrenza dei potenti produttori presenti da anni in questo settore, resero difficile la diffusione di Star7.

Il periodo successivo fu per questo quello più confuso e contraddittorio: alcuni dei creatori del progetto si impegnarono in una società per lo sviluppo e la diffusione della televisione interattiva, attività che finì per assorbire molte energie.

Nel 1994 gli enormi sviluppi del mercato del PC, unitamente alla tiepida accoglienza per la TV interattiva fecero comprendere che il PC era l'elettrodomestico di largo consumo che si cercava come piattaforma di lancio. Venne abbandonata così l'idea di utilizzare un hardware dedicato.

Siamo agli inizi del 1995 ed il progetto, sostenuto da Bill Joy, viene ribattezzato Live Oak: per battere la concorrenza e superare problemi di politica commerciale, si decise di rilasciare gratuitamente su Internet le specifiche del linguaggio e la piattaforma — sorgenti compresi. L'attività riprese vigore e in questa fase l'attenzione si focalizzò sulla parte del linguaggio, che presentava ancora smagliature e problemi da risolvere.

Nel 1996 nacque Java come linguaggio indipendente dalla piattaforma e fortemente orientato alla realizzazione di applicazioni per Internet.

L'ultima fatica del gruppo di lavoro prima dello scioglimento ufficiale del progetto fu quella di realizza una semplice applicazione e un ambiente in grado di interpretare l'HTML. Nacque così la prima Applet (la mascotte Duke che saluta) e il primo ambiente in grado di gestirla (il padre di HotJava). Il browser realizzato, applicazione semplice ma "reale", concretizzò finalmente, dopo anni di ricerche, il lungo e difficile processo di sviluppo che ha portato a Java.

Java il linguaggio portabile

Analizzando Java dal punto di vista del linguaggio si possono individuare alcune caratteristiche fondamentali: la prima e più importante è che il codice compilato (bytecode), senza necessità di ricompilazioni, è eseguibile su ogni tipo di piattaforma hardware-software che metta a disposizione una macchina virtuale Java. Dato che attualmente le virtual machines sono disponibili per ogni tipo di sistema operativo e per un numero molto elevato di hardware differenti, il bytecode è di fatto il primo esempio di portabilità reale e totale.

Java è un linguaggio a oggetti puro, dato che non è possibile programmare in modo non object oriented in parte o del tutto, come invece accade ad esempio con il C++.

La gestione della memoria, per motivi di sicurezza e di semplicità di programmazione, viene gestita dalla VM per mezzo di un efficiente Garbage Collector. Questo è forse uno degli aspetti più delicati di tutto il mondo Java e proprio le continue ricerche ed i progressi ottenuti hanno permesso alla piattaforma Java di diventare sempre più stabile e performante.

Una delle caratteristiche fondamentali di Java è che esso mette a disposizione un meccanismo di multithreading, col quale è possibile, all'interno della stessa applicazione, eseguire contemporaneamente più task.

La VM inoltre implementa un sistema automatico di loading delle classi in grado di caricarle in memoria — leggendole da disco o nel caso di applicazioni Internet scaricandole dalla rete — solo al momento dell'effettiva necessità, in maniera molto simile a quanto avviene con le DLL del sistema Windows.

Il linguaggio Java infine è stato progettato con il preciso obiettivo di offrire un elevato livello di sicurezza e semplicità d'implementazione; anche il debug di una applicazione richiede mediamente sforzi minori rispetto all'utilizzo di altre tecnologie.

L'eliminazione di alcuni costrutti complessi presenti invece in altri linguaggi come il C++, oltre ad offrire maggiori garanzie di funzionamento, ha avuto benefici ripercussioni anche sul processo di apprendimento. In tal senso una sintassi molto simile a quella del C++ rende il passaggio a Java meno problematico e sicuramente non così difficile come nel caso di un linguaggio totalmente nuovo.

Infine il livello di astrazione dato dall'introduzione di uno strato di software aggiuntivo svincola dal doversi preoccupare delle problematiche tipiche della piattaforma sottostante.

Sviluppo di applicazioni Java

Il problema delle performance è probabilmente il tallone di Achille dei programmi scritti in Java: l'utilizzo di codice bytecode interpretato, unitamente alla presenza di un software interpretato, fa si che un'applicazione Java mediamente sia meno efficiente di una scritta in codice nativo.

Questa osservazione è meno vera nel caso di applicazioni distribuite o fortemente orientate all'uso della rete. La semplicità con cui Java permette di costruire applicazioni net-oriented basate su architetture distribuite o multi-layer, consente di sfruttare a pieno i vantaggi di questi modelli di computazione e di ridurre l'importanza delle velocità assoluta.

Spesso le applicazioni Java, benché lente in senso assoluto, riescono ad essere ugualmente performanti se non migliori ad esempio di altre scritte in C/C++ utilizzando filosofie di progettazione tradizionali.

Inoltre, grazie alla compilazione al volo (JIT compiling) o alla possibilità di utilizzare ottimizzazioni dinamiche (come nel caso di HotSpot), le nuove virtual machine hanno ridotto notevolmente il gap fra Java ed altri linguaggi.

Le considerazioni sulla velocità di un programma perdono la loro importanza tenendo presente quelli che sono gli obiettivi principali di Java, e cioè portabilità, sicurezza, robustezza.

JDK

Sun, oltre ad aver definito questo nuovo linguaggio ha creato un set completo di API e una serie di tool per lo sviluppo di applicazioni Java. Il tutto, liberamente scaricabile da Internet, è contenuto nel Java Development Kit (JDK), che si evolve nel tempo a seconda delle innovazioni introdotte: al momento l'ultima versione è la 1.4

Le innumerevoli classi contenute nel JDK coprono tutte le casistiche di programmazione, dalla grafica alla gestione del multithreading, alla programmazione per la rete, alla manipolazione delle basi di dati.

Le classi sono raggruppate in packages secondo una organizzazione molto precisa. Attualmente è disponibile una quantità impressionante di classi e package aggiuntivi.

Ecco i package di base più importanti:
`java.lang`: le classi di base per poter sviluppare un'applicazione minima;
`java.util`: una serie di classi di pubblica utilità, dalla gestione delle funzioni matematiche, alle stringhe a strutture complesse (vettori, tabelle hash...);
`java.io`: supporto per l'I/O;
`java.net`: supporto per la programmazione di rete;
`java.awt`: mette a disposizione un set fondamentale di oggetti grafici per la creazione di GUI. I vari widget sono un subset di quelli utilizzati nelle varie piattaforme.
Questi sono stati i primi package introdotti con il JDK 1.0: una buona conoscenza dei sopraccitati è indispensabile per realizzare qualsiasi tipo di applicazione.

Applet Java

Si è detto che un programma Java può essere inserito in una pagina HTML e scaricato dinamicamente dalla rete. Il meccanismo, del tutto automatico, è effettuato dal classloader incluso nella virtual machine del browser.

Una Applet è quindi una applicazione che viene eseguita dalla macchina virtuale del browser. Essendo inserita in una pagina HTML anche se non è una prerogativa indispensabile, normalmente una Applet viene fornita di un'interfaccia grafica in modo da poterne controllare il funzionamento o visualizzare il risultato della sua esecuzione.

Le Applet non hanno particolari limitazioni rispetto a una normale applicazione, se non quelle imposte per motivi di sicurezza dal Security Manager della Virtual Machine: sotto queste condizioni una Applet non può accedere in nessun modo al file system della macchina locale, non può scrivere sul server da cui proviene e non può accedere a host diversi da quelli di provenienza.

Queste restrizioni, che possono sembrare limitare una qualsiasi attività della Applet, sono frutto di una precisa scelta volta a garantire un elevato livello di sicurezza; è questo un aspetto di fondamentale importanza per creare un sistema di network computing sicuro ed affidabile.

L'adozione di strutture client server o 3-Tier, consente di raggiungere un ragguardevole livello di flessibilità e potenza espressiva, senza violare i principi base della security.

Si tenga presente che, modificando la politica restrittiva del security manager, è possibile ridurre le restrizioni imposte, ottenendo applicazioni più potenti ma anche capaci di eseguire operazioni potenzialmente pericolose. Esistono in tal senso tecniche di certificazione delle Applet che permettono al browser di identificare l'Applet come fidata e di disattivare il controllo sulle operazioni effettuate quando viene mandata in esecuzione.

Tool del JDK

Come accennato precedentemente il Java Development Kit mette a disposizione, oltre alle librerie di base contenute nei vari packages, una serie di programmi per sviluppare e testate le proprie applicazioni Java.

Il comando `javac` manda in esecuzione il compilatore di bytecode. Deve essere invocato passando come argomento i nomi dei file da compilare. Ad esempio, supponendo di voler compilare una classe `HelloWorld`

```
C:\>javac HelloWorld.java
```

Come normalmente avviene in tutti i programmi invocabili da riga di comando, tramite l'utilizzo di appositi flag è possibile specificare le varie opzioni di compilazione: ad esempio definire l'inclusione delle informazioni per debug, o la directory dove sono contenute le classi di supporto. Ecco l'help prodotto dal compilatore se invocato senza parametri

```
C:\>javac
Usage: javac <options> <source files>
where possible options include:
-g                        Generate all debugging info
-g:none                   Generate no debugging info
-g:{lines,vars,source}    Generate only some debugging info
-O                        Optimize; may hinder debugging or enlarge class file
-nowarn                   Generate no warnings
-verbose                  Output messages about what the compiler is doing
-deprecation              Output source locations where deprecated APIs are used
-classpath <path>         Specify where to find user class files
-sourcepath <path>        Specify where to find input source files
-bootclasspath <path>     Override location of bootstrap class files
-extdirs <dirs>           Override location of installed extensions
-d <directory>            Specify where to place generated class files
-encoding <encoding>      Specify character encoding used by source files
-target <release>         Generate class files for specific VM version
```

Al termine del processo di compilazione si ottiene un file `.class` avente lo stesso nome della classe di partenza.

Il `javac_g` è la versione del compilatore che produce codice bytecode tale da facilitare l'eventuale debugging che si effettua con il comando `jdb`: questo tool è alquanto scomodo da utilizzare, non tanto per la modalità di esecuzione a riga di comando, ma piuttosto per la difficoltà dei comandi e delle istruzioni da eseguire. Questo fatto sembra confermare quanto detto da Brian Ritchie, secondo il quale il migliore debugging è quello che si può effettuare con l'istruzioni `println`.

Ogni applicazione compilata può essere mandata in esecuzione dall'interprete, eseguibile per mezzo del comando `java`. Riconsiderando l'esempio di prima si dovrà scrivere

```
C:\>java HelloWorld
```

Ecco l'output prodotto dall'interprete invocato senza parametri

```
C:\>java
Usage:   java [-options] class [args...] (to execute a class)
   or    java -jar [-options] jarfile [args..] (to execute a jar file)
where options include:
-cp -classpath <directories and zip/jar files separated by ;>
                          set search path for application classes and resources
-D<name>=<value>          set a system property
-verbose[:class|gc|jni]   enable verbose output
-version                  print product version and exit
-showversion              print product version and continue
-? -help                  print this help message
-X                        print help on non-standard options
```

Molto utile il flag `version` che permette di conoscere la versione del JDK correntemente in uso

```
C:\>java -version
java version "1.3.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0_01)
Java HotSpot(TM) Client VM (build 1.3.0_01, mixed mode)
```

Per poter compilare prima ed eseguire una classe, è necessario che siano impostate due variabili d'ambiente `path classpath`.

La prima, dal significato piuttosto intuitivo, consente di eseguire i comandi da console. La seconda invece informa il compilatore o l'interprete, dove sono installate le varie classi di sistema del JDK, ma anche tutte quelle necessarie per poter eseguire la compilazione e l'esecuzione. Ad esempio su piattaforma Windows si potrebbe scrivere

```
set CLASSPATH=".;C:\programs\java\mokapackages.jar;%CLASSPATH%"
```

in questo caso si aggiunge alla variabile d'ambiente variabile `CLASSPATH`, la directory corrente (piccolo trucco senza il quale molte volte si hanno malfunzionamenti apparentemente incomprensibili) e le classi contenute nell'archivio `mokapackages.jar`.

Se non si volesse modificare la variabile `CLASSPATH`, si può personalizzare il classpath direttamente al momento dell'invocazione del compilatore o dell'interprete tramite i flag `cp` o `classpath`. Questa seconda soluzione è da preferirsi solo se si hanno esigenze particolari (ad esempio volendo eseguire due programmi Java con due configurazioni differenti).

Normalmente il classpath di sistema, quello che indica dove sono installate i vari package del JDK, è impostato automaticamente al momento della installazione del JDK.

Nel caso si voglia eseguire una Applet è fornito un apposito interprete (appletviewer) al quale si deve passare il nome del file HTML nel quale è inserita l'Applet.

È possibile generare automaticamente un file di documentazione in formato HTML a partire dai commenti inseriti all'interno del codice Java. Ogni riga di commento inserita fra `/** e **/` viene ignorata dal compilatore ma non dal tool `javadoc`, che genera il file HTML. Nel commento del codice è quindi possibile inserire tag HTML come `<B>` o `<I>` per migliorare l'aspetto finale del documento. Caratteristica interessante è la possibilità di inserire tag non HTML (nel formato `@<XXX>`) che vengono interpretati dal `javadoc` per eseguire operazioni particolari. Ad esempio `@<see>` genera un link ad altre classi. Per maggiori approfondimenti sulla creazione di documentazione in Javadoc si veda l'appendice in fondo al libro.

Il tool `javah` permette di creare file header `.h` per far interagire classi Java con codice C. Infine il `javap` esegue un disassemblamento del bytecode restituendo il sorgente Java.

La prima applicazione

Come ultima cosa, al termine di questa panoramica introduttiva si può passare a vedere come generare una semplice applicazione Java. Senza troppa originalità si analizzerà una variante del classico HelloWorld. Nel caso di una applicazione, la versione minimale di questo programma potrebbe essere:

```
public class SalveMondo {

   public SalveMondo (){
      this.print();
   }

   public void print(){
      System.out.println("Ciao mondo");
   }

   public static void main(String[] args) {
      SalveMondo sm= new SalveMondo;
   }
}
```

Come si può notare la struttura di questo miniprogramma è molto simile a un equivalente scritto in C++. Per stampare nella console utente, si esegue chiamata al metodo `println` tramite lo stream di default `out` ricavato dalla classe `System`.
Supponendo utilizzare il sistema Windows, la compilazione potrà essere effettuata eseguendo da console la seguente istruzione

```
C:\>javac SalveMondo.java
```

mentre per l'esecuzione si dovrà eseguire a riga di comando il comando

```
c:\>java SalveMondo
```

La versione Applet dello stesso programma permetterà di visualizzare il risultato direttamente nella finestra del browser. In questo caso il testo verrà passato alla Applet come parametro esterno tramite tag `HTML`. Ecco il codice corrispondente

```
import java.applet.*;
import java.awt.*;

public class Salve extends Applet {
   String Text;
   public void init() {
      super.init();
      // si ricava il testo dal file HTML
      Text = new String(getParameter("text"));
   }

   public void paint (Graphics g) {
      g.drawString(Text,50,50);
   }
}
```

Nell'ordine si possono notare le seguenti parti: per creare una Applet si deve necessariamente implementare la classe `Applet`; non è necessario definire un `main()` poiché l'Applet parte automaticamente dopo aver lanciato il metodo `init` dove verranno poste tutte le istruzioni di inizializzazione della Applet; il testo da stampare viene prelevato dall'esterno come parametro della Applet, per mezzo del metodo `getParameter()` (di seguito è mostrato come deve essere strutturato tale file); la stringa non viene stampata ma disegnata nel metodo `Paint` per mezzo di una chiamata a `drawString()`.

Il file HTML deve essere così definito

```
<APPLET code="Salve.class" width=450 height=120>
<PARAM NAME=text value="Salve Mondo">
</APPLET>
```

L'Applet viene inserita e offerta al browser per mezzo del tag `<APPLET>`. Il keyword `code` specifica il nome del file `.class` contenente l'Applet. Nel caso di più file class, esso deve riferire a quello contente la classe principale (quella che deriva da Applet).

Per passare un parametro all'Applet si utilizza il tag `PARAM` seguito da `NAME` più il nome del parametro e da `VALUE` più il valore del parametro.

Prima di concludere è bene affrontare un esempio un poco più complesso che permetta di comprendere le effettive potenzialità del linguaggio. Si è pensato di utilizzare una Applet che mostrasse come realizzare una interfaccia grafica inserita in un browser.

Ecco il codice dell'esempio:

```
import java.awt.*;
import java.applet.*;

public class MyApplet extends Applet {
   java.awt.TextField textField1;
   java.awt.Checkbox checkbox1;
   java.awt.Button button1;
   java.awt.Label label1;

   public void init() {
      super.init();
      setLayout(null);
      addNotify();
      resize(453,358);
      textField1 = new java.awt.TextField();
      textField1.reshape(12,48,312,36);
      add(textField1);
      checkbox1 = new java.awt.Checkbox("Rosso");
      checkbox1.reshape(12,12,132,24);
      Font f= new Font("TimesRoman", Font.BOLD, 24);
      checkbox1.setFont(f);
      add(checkbox1);
      button1 = new java.awt.Button("Vuota");
      button1.reshape(336,48,84,36);
      add(button1);
      label1 = new java.awt.Label("text",Label.CENTER);
      label1.setFont(new Font("Courier",Font.BOLD, 35));
      label1.reshape(24,120,351,113);
      add(label1);
   }

   public boolean handleEvent(Event event) {
      if (event.target==textField1) {
         if (checkbox1.getState())
            label1.setForeground(new Color(255,0,0));
         else
            label1.setForeground(new Color(0,0,0));
         label1.setText(textField1.getText());
      }
      if (event.target==button1) {
         textField1.setText("");
         label1.setText(textField1.getText());
      }
```

```
   return super.handleEvent(event);
   }

}
```

Rispetto al caso precedente si noti la fase di inizializzazione dei vari componenti grafici e la loro successiva aggiunta al pannello principale per la visualizzazione.

Dopo aver creato gli oggetti e averli visualizzati, l'altra sezione importante è quella relativa alla gestione degli eventi, che sono intercettati dal metodo `handleEvent()`.

Per maggiori approfondimenti relativi alla gestione delle interfacce grafiche in Java, si rimanda ai capitoli dedicati ad AWT e Swing dove sono affrontate in dettaglio tali tematiche, analizzando in profondità sia il caso delle Applet che delle applicazioni stand alone.

Il linguaggio Java

di Giovanni Puliti

Introduzione

Java è un linguaggio orientato agli oggetti e, da questo punto di vista, l'impostazione data risulta piuttosto rigorosa, dato che non è possibile fare ricorso a organizzazioni miste del codice (in parte a oggetti, in parte strutturate) come avviene in altri linguaggi, primo fra tutti il C++.

La struttura sintattica obbliga inoltre ad adottare un formalismo ben preciso, che porta nella maggior parte dei casi a un codice elegante e ben strutturato.

Per questo motivo il programmatore che voglia addentrarsi nello studio del linguaggio, dovrebbe avere una buona conoscenza della teoria di progettazione e programmazione a oggetti.

L'obiettivo di questo capitolo non è di introdurre la programmazione a oggetti, per la quale si rimanda alla bibliografia riportata al termine, quanto piuttosto quello di mostrare i princìpi fondamentali della programmazione a oggetti in Java.

La programmazione a oggetti in Java

La filosofia Object Oriented di Java ha come obiettivo la creazione di un linguaggio estremamente semplice, facile da apprendere volto a eliminare quei costrutti pericolosi che portano in certi casi a situazioni non facili da gestire o prevedere. Ad esempio, oltre alla ben nota mancanza dei puntatori ad aree di memoria, in Java sono stati eliminati costrutti come l'ereditarietà multipla, i template e qualche altro elemento di minore importanza.

Oltre a semplificare il lavoro del programmatore, tali scelte sono orientate ad aumentare la sicurezza, intesa sia come protezione da errori accidentali che come prevenzione nei confronti di operazioni ostili da parte di estranei.

Classi, variabili e metodi

Una classe in Java si definisce per mezzo della parola chiave `class`. Nell'esempio che segue si può notare già una certa somiglianza con il linguaggio C++:

```
public class MiaClasse {
   public int variabileIntera;
   void mioMetodo() {
      // fai qualcosa
   }
} // fine classe
```

Per descrivere il corpo della classe, all'interno delle parentesi graffe si definiscono i vari metodi, variabili membro, e in alcuni casi particolari anche altre classi (dette inner classes).

Uno degli elementi fondamentali della programmazione a oggetti è la visibilità fra classi differenti o appartenenti a packages (di cui si parlerà in seguito); in questo caso la parola chiave `public` indica visibilità totale per la classe da parte di altre porzioni di codice. Gli specificatori di accesso ammessi in Java sono quelli corrispondenti alle keyword `public`, `protected`, `private` e default al quale però non corrisponde nessuna parola chiave. Più avanti verranno affrontate le regole base per la visibilità, con il significato di queste parole.

Per quanto riguarda le regole di denominazione esistono alcune indicazioni di massima e delle norme ben precise da seguire. La regola fondamentale è che in ogni file deve esistere al massimo una classe pubblica: questo non significa che debba essere l'unica, ma è obbligatorio che sia l'unica con specificatore `public`.

A volte si dice che le classi non pubbliche inserite nello stesso file della classe pubblica, essendo visibili solo all'interno del package, svolgono il compito di classe di servizio, mettendo a disposizione alcuni metodi di utilità per la classe pubblica del file.

Nel caso in cui non si disponga di una buona conoscenza della programmazione a oggetti, e dei pericoli legati a una non corretta analisi, si consiglia di limitarsi ad una sola classe per file, indipendentemente dalla sua visibilità. Questa indicazione diventa ancora più importante nel caso in cui si utilizzi una organizzazione delle classi basata su package.

Per quanto riguarda le regole di naming delle variabili e metodi, esiste un formalismo adottato da Sun (si veda [codeconventions]): molto brevemente si può dire che tutti nomi di classi devono iniziare per lettera maiuscola (così come le lettere iniziali delle parole composte), mentre i nomi di variabili e di metodi devono seguire la forma canonica (prima lettera minuscola e per le parole composte le iniziali maiuscole). Ad esempio

```
NomeClasse                  nomeMetodo()                  nomeVariabile;
```

Questo formalismo è puramente indicativo e serve, tra l'altro, per distinguere il codice Java da quello basato su formalismo COM o altro. Solo in alcuni casi, come ad esempio

nei JavaBeans o nella estensione di particolari interfacce da parte di classi, è obbligatorio seguire alla lettera tale schema.

Strutturazione del codice: ereditarietà, implementazione di interfacce

Come noto uno dei fondamenti base della programmazione a oggetti, insieme al polimorfismo e all'incapsulamento, è l'ereditarietà. Per specificare che la classe A eredita o deriva da B si utilizza la parola chiave `extends`; ad esempio:

```
public class A extends B {
   ...
   corpo della classe A
   ...
}
```

È importante notare che in Java ogni classe in cui non sia definito esplicitamente un padre, eredita automaticamente dalla classe `Object` (è il compilatore che introduce automaticamente questa relazione).

Questo fatto influenza pesantemente tutta la filosofia di base della programmazione in Java dove si avrà sempre una sola gerarchia di classi facenti capo alla classe `Object`. Volendo utilizzare una metafora, in Java si ha un solo grande albero, e non tanti cespugli come ad esempio accade in C++, dove ogni gerarchia di oggetti è slegata da un'altra.

Anche se il suo significato è fondamentalmente differente, accanto a `extends` troviamo la keyword `implements` che permette a una classe di implementare una determinata interfaccia.

La sintassi che si utilizza è la seguente:

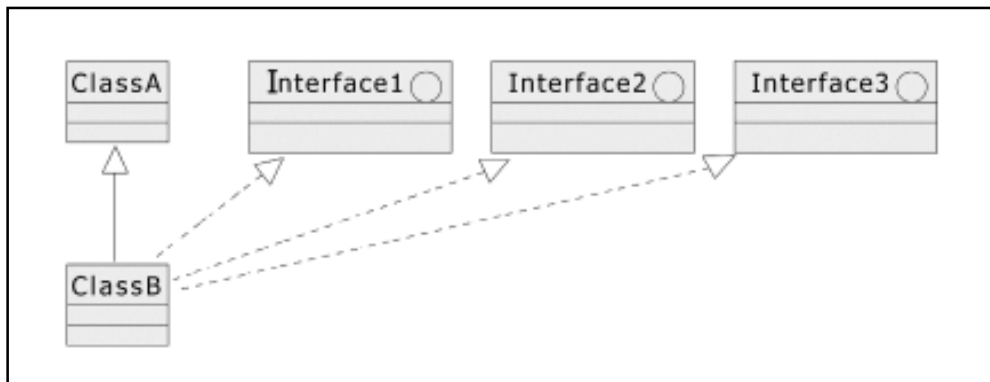
```
public class ClassB extends ClassA implements Interface1, Interface2, ... {
   ...
   corpo della classe A
   ...
}
```

dove `Interface1`, `Interface2`, `Interface3`, … sono le interfacce da implementare.

A questo punto sorge spontaneo il dubbio su che cosa sia e a cosa serva una interfaccia, e quale sia la differenza rispetto al costrutto di classe. Senza entrare in aspetti strettamente legati alla teoria della programmazione a oggetti, si dirà qui brevemente che una interfaccia è uno schema da seguire ogni qual volta si desideri aggiungere una serie di funzionalità a una determinata classe.

Dato che in Java è stata eliminata l'ereditarietà multipla, l'utilizzo delle interfacce permette di realizzare strutture gerarchiche più o meno complesse pur mantenendo la regola di base "un solo padre per ogni figlio".

Per maggiori approfondimenti sui vantaggi e sull'importanza di utilizzare le interfacce si veda l'appendice "Programmare con le interfacce" al termine del capitolo.

Figura 2.1 – *La regola di base della ereditarietà in Java impone che ogni classe possa estendere al massimo una classe, anche se può implementare un numero qualsiasi di interfacce.*

I packages

L'organizzazione delle classi in gerarchie di oggetti è un fatto ormai consolidato in ogni linguaggio a oggetti, mentre l'utilizzo dei package, sebbene non sia prerogativa di Java, ha avuto nuovo impulso grazie a questo linguaggio.

Un package è un particolare contenitore che può raccogliere al suo interno una serie di classi o altri packages: strutturalmente seguono il modello delle directory del filesystem, le quali possono contenere sia file che altre directory.

Fra i vari motivi che hanno spinto Sun ad adottare una organizzazione a package, vi è la volontà di semplificare e ordinare il codice. Se si pensa infatti alle dimensioni raggiunte dal JDK, questa necessità è quanto mai importante: dal JDK 1.0 che comprende 212 fra classi e interfacce e 8 packages, si è passati al JDK 1.1 che comprende 504 classi/interfacce e 23 packages. Il JDK 1.2, che comprende 1520 classi/interfacce e 59 packages, con le standard extensions raggiunge la ragguardevole cifra di 2000 classi/interfacce raccolte in 98 packages.

Utilizzo dei package

Più classi possono essere impacchettate logicamente in un unico contenitore, un package, e in tal caso esse vengono individuate dall'indicazione `nomepackage.nomeclasse`.

L'organizzazione delle classi secondo una logica a package comporta il salvataggio dei file corrispondenti in una struttura gerarchica delle directory isomorfa a quelle dei package aventi i medesimi nomi: in pratica se una classe è organizzata secondo la struttura

```
com.mokabyte.util.NomeClasse
```

allora dovrà essere salvata in una directory secondo la struttura

```
com/mokabyte/util/
```

Nel caso in cui si abbia bisogno di importare alcune classi esterne (esterne alla propria classe o al package in cui si sta lavorando) si usa la direttiva `import` con il nome del package da importare, in maniera analoga alla direttiva `#include` del C++.

La suddivisione logica e il raggruppamento di classi affini permette una miglior organizzazione del codice e in particolare impedisce i conflitti tra nomi uguali (name clashes): per esempio due classi di nome `Point` possono coesistere in due package diversi, senza che questo comporti nessun problema in fase di compilazione o di utilizzo.

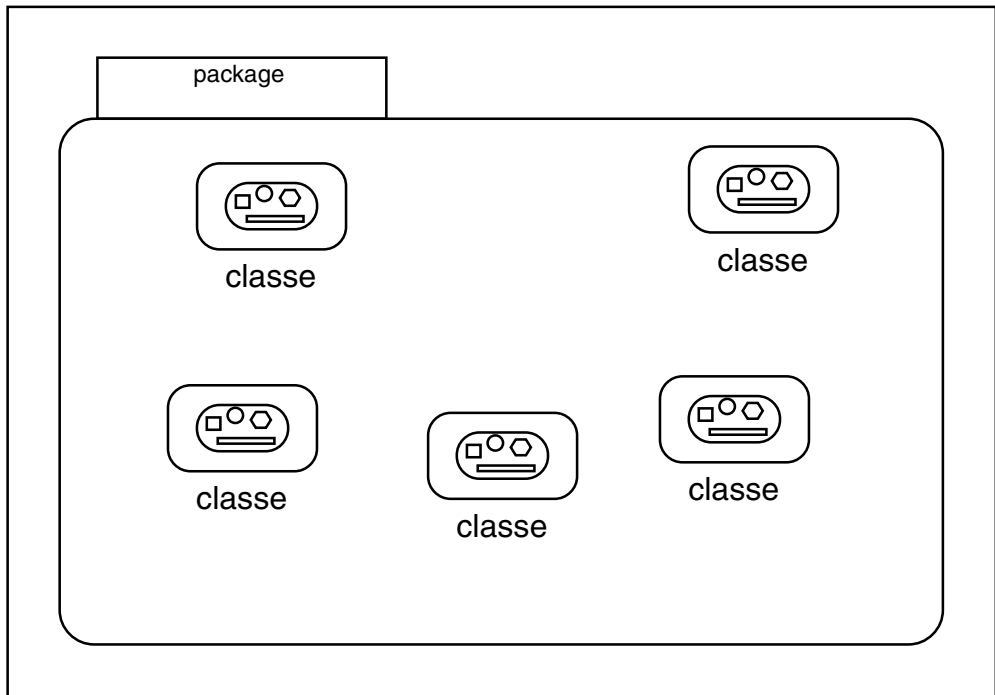
Questo garantisce un maggior ordine e riuso del codice: due package diversi possono sempre essere utilizzati contemporaneamente da un unico programma. A una più attenta analisi si nota che in realtà il problema dei conflitti di nomi si ripresenta, seppur semplificato, nella scelta del nome del package. La proposta di Sun è la seguente: far precedere a ogni nome di package il dominio Internet invertito dell'ente per cui si lavora. Per esempio, tutto il software scritto all'interno di un certo gruppo di lavoro Sviluppo Web (SW), facente parte della divisione Progettazione e Sviluppo Software (PSS) nella azienda MokaByte dovrebbe essere confezionato in package il cui nome inizia con `com.mokabyte.pss.sw`, dato che il dominio dell'azienda è `mokabyte.com`.

L'unicità dei nomi di package viene così garantita dall'unicità dei nomi di Internet, automaticamente, senza preoccuparsi di dover definire alcuna autorità di controllo centralizzata. In realtà il significato di questa scelta è ancora più sottile: nello spirito di quello che sarà il network computing, il codice dovrebbe essere riutilizzato il più possibile e distribuito via rete ogni volta che viene richiesto. Se qualcuno nel mondo scriverà un programma che utilizza un package scritto dal team di sviluppo di cui sopra, non sarà necessariamente costretto a ridistribuirlo con il suo codice, ma potrà semplicemente importarlo con il suo nome completo: `com.mokabyte.pss.sw.nomepackage`.

Ovviamente l'azienda che mette a disposizione tale software dovrà configurare un server in grado di distribuire tali package su richiesta.

Il nome completo rimane quindi codificato nel codice eseguibile, così le virtual machine Java delle prossime generazioni saranno in grado di risalire al nome del server in grado di distribuire il package richiesto e quindi di ricuperarlo via rete anche se vengono eseguite a Giringiro, Australia, da Mr. Crocodile Dundee...

In pratica le cose sono un po' più complesse e richiedono che molti altri dettagli di questo meccanismo siano definiti. Si ricordi infine che è stato adottato lo schema `com.mokabyte.mokabook.`*`nomecapitolo`* per organizzare gli esempi allegati a questo libro.

Figura 2.2 – *La strutturazione del codice in package permette di raggruppare le varie classi in funzione del loro scopo.*

Per creare un proprio package è necessario utilizzare la keyword `package` come prima riga del file `.java`. Più precisamente se si desidera creare una classe pubblica A che appartenga al package `mypackage`, allora sarà necessario scrivere all'interno del file `A.java`

```
package mypackage;

public class A {
   ...
   corpo della classe
   ...
}
```

In questo modo la classe generata apparterrà al package che verrà creato automaticamente. In questo caso, al momento della compilazione verrà generata una directory denominata `mypackage` ed il file `A.class` verrà automaticamente posizionato in tale directory. Nel caso in cui si voglia creare un ulteriore package (denominato `mypackage2`) annidato all'interno di `mypackage`, si dovrà scrivere

```
package mypackage.mypackage2;

public class A2 {
   ...
   corpo della classe
   ...
}
```

Analogamente al caso precedente, la classe generata dal compilatore verrà posizionata nella directory `mypackage/mypackage2`. L'utilizzazione di package per semplificare l'organizzazione delle classi, già di per sé molto utile, diviene preziosissima se non inevitabile in progetti complessi. Molto spesso però il programmatore alle prime armi si trova in difficoltà nel cercare di utilizzare tale organizzazione delle classi. La domanda più comune che sorge spontanea è, per esempio, quella relativa al posizionamento dei sorgenti. Se infatti è automatica la collocazione del `.class` all'interno della directory corrispondente, dove si deve posizionare il file sorgente `.java` e da dove si deve invocare la compilazione? La scelta più semplice che in genere si consiglia — sia che si utilizzi un compilatore da riga di comando come il javac contenuto nel JDK, sia che si sviluppi un progetto per mezzo di un ambiente evoluto come ad esempio *JBuilder* — è quella di generare prima la struttura gerarchica delle directory e di copiare poi i sorgenti nella posizione opportuna, seguendo la struttura dei vari package.

In questo modo se dalla classe A che appartiene al package `mypackageA` si desidera fare riferimento alla classe B contenuta in `mypackageB` sarà sufficiente creare le directory `mypackageA` e `mypackageB` e copiare i file `A.java` nella prima e `B.java` nella seconda.

Per permettere alla classe B di vedere la classe A e non ricevere nessun errore in fase di compilazione è sufficiente definire la classe B

```
package mypackageB;

import mypackageA;

public class B {
   // si utilizza la classe A
   A a;

   public B() {
      a = new A();
   }
}
```

Infine per non incappare in messaggi di errore da parte del compilatore si deve fare attenzione ad altre due semplici ma importantissime indicazioni: la prima è che, quando si utilizza una organizzazione a package, per identificare univocamente una determinata classe è necessario specificare il nome assoluto, ovvero `package.NomeClasse`.

Questo risulta essere piuttosto ovvio nel momento in cui si utilizza una classe contenuta in qualche package, dovendo scrivere

```
// all'inizio del file
import mypackage.mypackage2.A
// da qualche parte nel codice
A a = new A();
```

Quello che invece è meno evidente è come effettuare la compilazione della classe A: dato che il `package.NomeClasse` identifica univocamente la classe, è necessario invocare il compilatore dalla directory radice rispetto al package stesso. Questo significa che nel caso in esame, si dovrà invocare il comando

```
javac mypackage.A
```

dalla directory che contiene la directory `mypackage`, ovvero due livelli sopra quella che contiene la A.

Questo aspetto non si presenta a chi utilizza un tool di editing, ed è per questo che in genere se ne sconsiglia l'uso a coloro che si avvicinano a Java per la prima volta, preferendo il classico JDK che richiede una maggiore attenzione e quindi consente di comprendere a fondo certi meccanismi di base, come quello dei package.

L'altro piccolo trucco che molto spesso fa risparmiare ore nella ricerca di un bug inesistente riguarda l'errato settaggio della variabile `classpath`: si ricordi infatti che per far sì che una classe venga trovata dal compilatore, è necessario che essa si trovi in una directory puntata dalla variabile d'ambiente di cui sopra.

Quello di cui spesso non si tiene conto è che, anche se si trova nella directory corrente, la classe non verrà trovata dal compilatore, a meno di non aver specificato nel `classpath` di includere la directory corrente. Ad esempio si potrebbe scrivere

```
classpath = ".;c:\java\jdk1.1\lib\classes.zip;"
```

con particolare attenzione a "`.`".

Infine è da notare che mentre il codice realizzato in proprio può essere organizzato o meno con una struttura a package, le classi del JDK seguono pesantemente tale schema. I package del JDK più importanti sono essenzialmente:

`java.lang`: classi base del linguaggio, delle quali non è necessario effettuare l'importa-

zione essendo automatica (`Object`, `Thread`, `Throwable`, `System`, `String`, `Math`, wrapper classes, ...)

`java.io`: classi per la gestione dell'I/O (`FileInputStream`, `FileOutputStream`)

`java.util`: classi di utilità (`Date`, `Random`, ...)

`java.net`: classi di supporto alle applicazioni di rete (socket, URL, ...)

`java.applet`: classe `Applet`, ...

`java.awt`: Abstract Windowing Toolkit

Si tenga presente che in Java non esiste il termine di libreria, utilizzato nei linguaggi procedurali non a oggetti. Infatti secondo una delle regole base della OOP, una classe ha un comportamento ma anche uno stato: il termine libreria sarebbe riduttivo, facendo convenzionalmente riferimento a raccolte di funzioni.

Eccezione fanno quei pochi casi, per esempio quello della classe `java.lang.Math` che è composta da metodi statici, per cui può essere assimilata a una raccolta di routines che non potevano essere collezionate all'esterno di una classe.

In genere le collezioni di classi organizzate in package sono poi incluse in un archivio unico in formato ZIP o JAR (Java Archive).

Chi volesse verificare, può aprire i vari `.zip` o `.jar` del JDK e controllarne il contenuto: si tratta di collezioni di classi che una volta scompattate danno luogo a strutture a directory identiche alla struttura gerarchica dei package. Da qui la conferma dello stretto legame fra directories e packages.

Access control

In Java esistono 3 specificatori di accesso: `public`, `private e protected`, con i quali si possono specificare l'accessibilità di una classe, metodo o variabile, secondo le seguenti specifiche

`public`: una proprietà o un metodo all'interno di una classe A che venga definito pubblico acquista visibilità totale. Potrà infatti essere visto al di fuori della classe, sia dalle classi che ereditano dalla classe A, sia da quelle classi chi non estendono A. Anche le classi esterne al package di A potranno accedere liberamente a tali elementi.

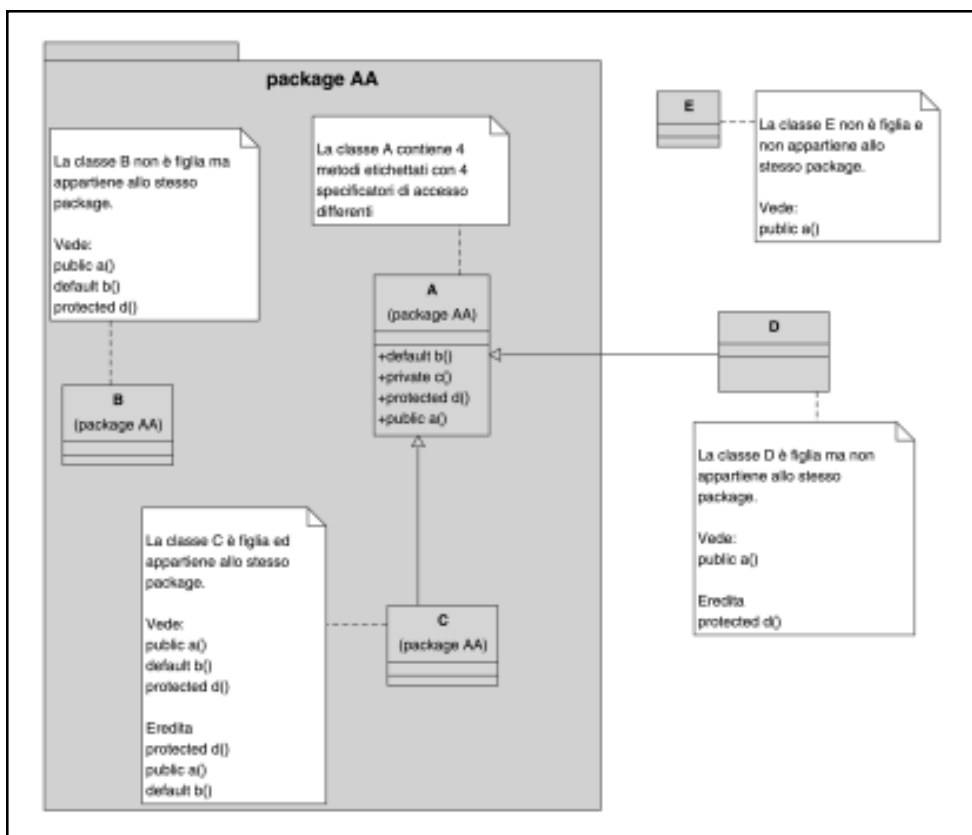
`private`: questo specificatore è di fatto il simmetrico di public, dato che blocca ogni tipo di visibilità. Metodi o proprietà della classe A dichiarati privati hanno visibilità limitata

alla sola classe A, nemmeno le classi derivate potranno accedervi.

`protected`: i metodi e le proprietà dichiarate protette potranno essere accedute da altre classi esterne solo se queste apparterranno allo stesso package. All'esterno del package nessuno potrà accedere a tali elementi, mentre esclusivamente le classi figlie li erediteranno al loro interno.

Esiste inoltre una ulteriore modalità di accesso, detta default o automatic: a questo caso non corrisponde una parola chiave, trattandosi del livello di visibilità che viene assegnato automaticamente nel caso in cui non venga specificata esplicitamente nessuna modalità. In questo caso la visibilità è limitata alla gerarchia di classe e al package.

Nella figura 2.3 sono rappresentate in modo schematico le regole appena viste.

Figura 2.3 – *Regole di visibilità all'interno di gerarchie e packages.*

A differenza di quanto avviene in C++, in Java non esiste la possibilità di specificare la visibilità associata alla relazione di specializzazione. Se ad esempio abbiamo che B estende A, avremo sempre che, da un punto di vista C++, tale relazione è pubblica. Questo significa che tutti i metodi/proprietà visibili da B manterranno lo stesso livello di visibilità.

In Java quindi non è possibile scrivere un'espressione del tipo

```
public class B extends private A
```

Infine i soli specificatori di accesso assegnabili alla classe completa sono `public` e `default` (nessuno specificatore). Nel primo caso la classe è pubblica, e perciò può essere vista da qualsiasi posizione sia all'interno del package o della gerarchia, che all'esterno. Si tenga presente che in un file `.java` si può definire al più una classe pubblica.

Una classe senza nessuno specificatore invece limita la propria visibilità al solo package.

Gli altri specificatori: `static` e `final`

Le keyword `public`, `private` e `protected` non sono le uniche che si possono utilizzare in abbinamento a un metodo o a una classe per specificarne ulteriormente la natura.

Esiste anche `static`, che ha un significato completamente differente, e varia a seconda che si riferisca a un metodo o a una proprietà.

Nel caso si definisca una proprietà `static`, allora tale proprietà sarà condivisa fra tutte le istanze di tale classe. Ad esempio, supponendo che sia

```
public class MyClass {
   static int Stat;
...
}

MyClass Mc1 = new MyClass();
MyClass Mc2 = new MyClass();
MyClass Mc3 = new MyClass();
```

ed istanziando la variabile `Stat` del reference `Mc1`

```
Mc1.Stat = 10;
```

allora anche le corrispettive variabili delle istanze `Mc2` e `Mc3` conterranno il valore 10.

Lo specificatore `static` applicato invece a un metodo indica che esso può essere invocato anche se la classe di cui fa parte non è stata istanziata.

Un caso tipico è quello del metodo `main` che può — e anzi deve — essere chiamato senza che la classe sia stata creata con una chiamata a `new`: in questo caso non potrebbe essere altrimenti, dato che tale metodo ha proprio lo scopo di far partire l'applicazione, quando nessuna classe ancora esiste.

È importante notare che dall'interno di un metodo statico nasce il concetto di contesto statico (*static context*): in quest'ottica da un contesto statico è possibile invocare esclusivamente metodi statici ed accedere a variabili definite anch'esse in un contesto statico. Le altre infatti potrebbero non esistere nel caso in cui la classe non sia stata instanziata.

Tutti i metodi possono essere definiti statici tranne, per ovvi motivi, il costruttore di classe.

Il metodo statico può quindi essere invocato sulla classe e non necessariamente sull'oggetto: ad esempio avendo

```
Public class MyClass {
   Public static myMethod() {}

   Public static void main(String args[]) {
      MyClass.myMethod();
   }
}
```

Anche la key `final` assume un significato differente a seconda che venga abbinata a un metodo o a una proprietà. In questo secondo caso infatti serve per definire un valore costante che non può più essere modificato dopo la definizione e che quindi, proprio per questo motivo, deve essere specificato al momento della definizione/creazione. Ad esempio

```
public final int i=10;
```

Invece un metodo `final` indica che tale metodo non può più essere ridefinito (override) nelle eventuali classi figlie.

Parole chiave

In Java le parole chiave da utilizzare sono

```
abstract      double        int           super
boolean       else          interface     switch
break         extends       long          synchronized
byte          final         native        this
case          float         package       throw
char          for           private       throws
class         (goto)        protected     transient
(const)       if            public        try
continue      implements    return        void
default       import        short         volatile
do            instanceof    static        while
```

La maggior parte di esse molto probabilmente risulterà ben nota anche per programmatori alle prime armi con Java. In alcuni casi invece si tratta di parole legate a costrutti avanzati di programmazione, che saranno affrontati nel prosieguo del capitolo.

Per il momento si ponga attenzione alle parole relative ai tipi (`int`, `long`, …), a quelle che specificano il controllo del flusso (`if`, `for`, `while`, …), quelle relative alla programmazione a oggetti (come ad esempio `extends` ed `implements` ma anche `protected`, `final`, o `package`), quelle relative alla gestione delle eccezioni (`try`, `catch`, `throw` e `throws`)

Si noti infine che le chiavi `goto` e `const` pur non facendo parte del set di parole chiave utilizzabili, compaiono fra le keyword: in questo caso infatti il linguaggio se ne riserva l'uso pur non utilizzandole.

Controllo del flusso

Per controllare il flusso delle operazioni all'interno dei metodi, Java mette a disposizione una serie di parole chiave e di costrutti molto simili a quelli che si trovano negli altri linguaggi di programmazione procedurali.

In particolare i vari `if-then-else`, `for` e `while` si basano su una sintassi molto semplice che ricalca più o meno fedelmente quella messa a disposizione dal linguaggio C. Di seguito è riportato un breve promemoria della sintassi e del significato di ogni costrutto.

Costrutto `if else`

```
if(espressione booleana)
istruzione1

else
istruzione2
```

Esegue il blocco *`istruzione1`* se *`espressione booleana`* assume il valore booleano `true`, altrimenti esegue il blocco *`istruzione2`*.

Costrutto `while` e `do while`

```
while(espressione booleana)
   istruzione
```

Esegue il blocco *`istruzione`* fino a che *`espressione booleana`* assume il valore booleano `true`.

Funzionalità analoga è offerta dal costrutto `do-while`

```
do
istruzione
while(espressione booleana)
```

La differenza fra i due costrutti è che nel primo caso il controllo viene effettuato prima di eseguire istruzione, mentre, nel secondo caso, dopo.

Costrutto `for`

```
for(espressione di inizializzazione; espressione booleana;
    espressione di incremento) {
   istruzione
}
```

Esegue il blocco `for` fino a che *`espressione booleana`* assume valore booleano `true`; la *`espressione di inizializzazione`* viene eseguita solo la prima volta, mentre *`espressione di incremento`* ad ogni iterazione.

Costrutto `switch`

```
switch variabile
   case valore1:
      istruzione1
   case valore2:
      istruzione2
   case valore3:
      istruzione3
   case valore4:
      istruzione4
default:
```

Esegue un controllo sulla variabile intera *`variabile`* ed esegue il blocco di codice che segue il `case` corrispondente al valore assunto dalla variabile. Infine viene sempre eseguito il blocco `default`.

Tale costrutto prevede che l'esecuzione del blocco relativo al `case` *n* non impedisca l'esecuzione dei blocchi successivi. Per imporre invece tale restrizione si utilizza la parola chiave `break`, la quale inoltre viene utilizzata genericamente per uscire da un qualsiasi blocco di codice (dall'interno del corpo di un metodo come da un ciclo `for` o `while`).

Costrutto `continue`

Un'espressione `continue` provoca il salto alla fine del ciclo dove è inserita, passando direttamente alla valutazione della espressione booleana che controlla tale ciclo.

In un ciclo è possibile specificare una etichetta di un ciclo più esterno, in modo da applicare la funzionalità `continue` a tale ciclo e non a quello interno. Si tenga presente che in Java non è presente il `goto`: una delle funzioni di tale costrutto, presente in altri linguaggi procedurali, è quella ad esempio di controllare cicli esterni a quello annidato, cosa che può essere fatta con il `break` e `continue` con etichetta. Altro scopo del `break` è quello di saltare un blocco di codice (non interno a un ciclo) in concomitanza di una qualche condizione di errore: anche in questo caso il `break` con etichetta può essere utilizzato in sostituzione.

Infine un ulteriore impiego del `goto` è quello che consente di eseguire codice di pulizia prima dell'uscita da un metodo o da un blocco: anche in questo caso in Java si utilizza il `break` con etichetta o il costrutto `finally` se si è all'interno di una `try-catch`.

Costrutto `return`

Viene utilizzato per permettere a un metodo di restituire un valore all'esterno. Ad esempio:

```
public int add(int i1, int i2) {
   return i1 + i2;
}
```

`return` deve restituire un valore assegnabile a quello definito nella firma del metodo: la regola base da seguire è quella utilizzata normalmente in tutte le operazioni di assegnazione, quindi se il tipo non è esattamente lo stesso, deve essere effettuato un cast implicito o forzato. Si veda a tal proposito la parte relativa ai tipi riportata poco più avanti.

Per quanto riguarda il costrutto `try-catch` si veda la sezione dedicata alla gestione delle eccezioni.

Definizione/invocazione di metodi e passaggio di parametri

Una delle caratteristiche fondamentali di Java è che i parametri nelle invocazioni dei metodi sono passati per valore. Questa modalità, come ben noto è una delle due alternative possibili utilizzabile insieme a quella per riferimento nella maggior parte dei linguaggi procedurali: essa si traduce nella realizzazione di una copia del parametro tutte le volte che si effettua una chiamata a un determinato parametro.

Ad esempio, nel caso in cui si abbia

```
Public void myMethod (String str) {
   ...
   fai qualcosa con il parametro str
}
```

e supponendo di invocare tale metodo nel seguente modo

```
...
String stringa = "casa";
myMethod(casa);
```

allora dentro il metodo `myMethod()` verrà utilizzata una copia della variabile e le modifiche effettuate all'interno di tale metodo non verranno viste dall'esterno.

Questo è ciò che formalmente accadrebbe sulla base della modalità di passaggio dei parametri per valore. In realtà le cose non stanno esattamente in questi termini.

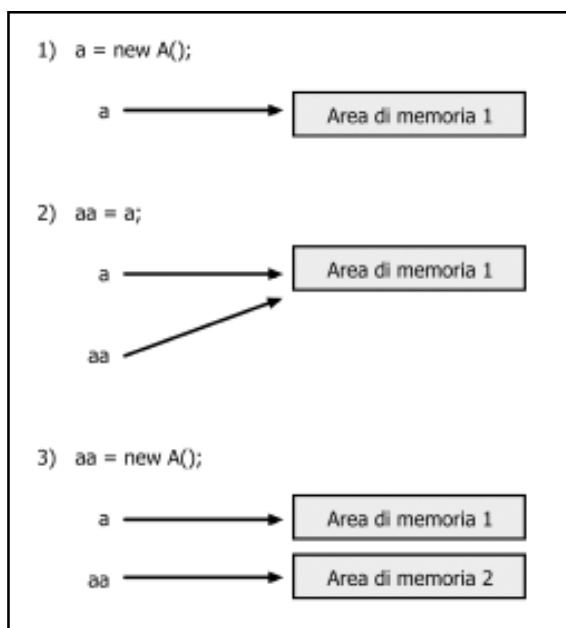
Infatti la copia di cui sopra non è esattamente quella della area di memoria che contiene la stringa “casa”, ma piuttosto del reference che punta a tale area di memoria (di fatto è come se in linguaggio C effettuassimo una copia del puntatore all’area di memoria ma non dell’area vera e propria).

Dovrebbe essere piuttosto chiaro quindi che, utilizzando tale reference per modificare il contenuto della cella dall’interno del metodo, si otterrà una modifica visibile all’esterno anche se abbiamo utilizzato una copia del reference (proprio per quanto mostrato nella figura 2.4).

Quindi il risultato finale dipende dal tipo di modifica che si effettua con il reference: se infatti da dentro il metodo si scrivesse

```
Str = "auto";
```

si otterrà solamente di deviare il puntatore copia a un’altra area di memoria lasciando inalterata quella che conteneva la stringa “casa”.

Figura 2.4 – *Quando si passa una variabile come parametro nella invocazione di un metodo, il runtime effettua una copia del reference che però punta alla stessa area di memoria originale. Se si effettua una qualsiasi operazione di assegnazione al nuovo reference, allora si otterrà un disaccoppiamento fra i due reference che punteranno effettivamente ad aree di memoria differenti.*

La cosa invece sarebbe completamente differente se si utilizzasse un ipotetico metodo (che però non è presente nella `String` ma nella `StringBuffer`) che agisce direttamente sull'oggetto memorizzato, come ad esempio

```
Str.append("auto")
```

Quindi accedendo all'oggetto con metodi che operano sul contenuto dell'oggetto si può modificare l'originale anche se si utilizza un reference copia.

Più formalmente si può quindi dire adesso che in Java i parametri sono passati per copia, anche se la copia è limitata al reference e non al contenuto.

Ovviamente, nel caso di tipi primitivi, il processo è molto più semplice, dato che viene effettuata realmente una copia del parametro.

Overloading e overriding

La relazione generalizzazione–specializzazione è una delle colonne portanti della programmazione a oggetti, e di fatto si basa sul concetto di ereditarietà fra classi. Strettamente legata a questo concetto vi è la possibilità di definire più versioni dello stesso metodo: le due modalità con cui si può operare sono dette overloading e overriding di metodi.

Nel primo caso questo significa che, dato ad esempio un metodo definito nel seguente modo

```
// versione 1
final void myMethod() {
   ...
   fai qualcosa
   ...
}
```

è possibile definire più versioni variando il numero e il tipo dei parametri, e/o la visibilità. Ad esempio si potrà scrivere anche

```
// versione 2
final void myMethod(String str) {
   ...
fai qualcosa
   ...
}

// versione 3
final void myMethod(int i) {
   ...
   fai qualcosa
   ...
}
```

Non è detto che questa *sovrascrittura* debba essere presente all'interno della stessa classe, ma può avvenire anche all'interno delle classi derivate.

È importante notare che i tre metodi appena visti sono effettivamente tre metodi differenti, e la scelta verrà fatta in funzione della chiamata: il runtime infatti effettua una operazione di match sulle differenti versioni del metodo e manda in esecuzione quello la cui firma corrisponde con quella della chiamata.

Ad esempio scrivendo

```
myMethod
```

verrà eseguito il primo, mentre nel caso di

```
myMethod(5)
```

verrebbe eseguito il terzo.

È molto importante notare che, similmente a quanto avviene in altri linguaggi come il C++, il compilatore non è in grado di distinguere fra due versioni dello stesso metodo se la differenza è solamente nel tipo ritornato: in altre parole i due metodi

```
// versione 2.a
final void myMethod(String str){
   ...
   fai qualcosa
   ...
}

// versione 2.b
final String myMethod(String str){
   ...
   fai qualcosa
   ...
}
```

i quali differiscono solamente per il tipo ritornato, sono visti come lo stesso, e per questo viene generato un errore in fase di compilazione. Discorso analogo per le eccezioni che un metodo può lanciare per mezzo della clausola `throws` di cui si parlerà in seguito.

L'altra modalità di ridefinizione dei metodi è quella detta di overriding; prima di vedere che cosa significhi, si consideri il caso in cui la classe base sia così definita

```
public class MyClass1 {
   final void myMethod() {}
}
```

mentre quella derivata sia

```
public class MyClass2 extends MyClass1 {
   ...
}
```

Allora in fase di esecuzione, supponendo di definire prima

```
MyClass1 Mc1 = new MyClass1();
MyClass2 Mc2 = new MyClass2();
```

le due invocazioni

```
Mc1.myMethod();
Mc2.myMethod();
```

porterebbero a un risultato del tutto identico, dato che in entrambi i casi verrebbe invocato il metodo della classe `MyClass1`.

Se invece si modifica la classe figlia per mezzo di una operazione di overloading del metodo `myMethod`, ovvero se si opera un cambiamento nel codice nel seguente modo

```
public class MyClass2 extends MyClass1 {
   // non permesso dal compilatore
   final void myMethod() {
   }
}
```

allora si avrà un comportamento differente, dato che infatti l'istruzione

```
Mc1.myMethod();
```

manderà in esecuzione il metodo della classe padre, mentre

```
Mc2.myMethod();
```

manderà in esecuzione quello della classe figlia. Questa caratteristica è molto importante nel caso si vogliano implementare comportamenti dinamici che in genere cadono sotto la definizione di comportamenti polimorfici di una classe o, più genericamente, di polimorfismo. Questi aspetti saranno esaminati più avanti nella sezione dedicata al polimorfismo in Java.

I costruttori di classe

Il costruttore di una classe è un particolare metodo che serve per creare una istanza della classe stessa. L'invocazione avviene per mezzo della keyword `new`, come ad esempio

```
String str = new String();
```

A differenza di altri linguaggi in Java non è possibile definire e quindi utilizzare il distruttore, anche se esiste un metodo particolare, detto `finalize`, che sarà esaminato più avanti.

Per definire il costruttore della classe è necessario e sufficiente creare un metodo con lo stesso nome della classe.

Ad esempio, se si ha una classe detta `MyClass`, il costruttore avrà questa forma

```
// Costruttore di default
public MyClass() {}

// Un altro costruttore
public MyClass(String str) {}
```

Il primo dei due viene detto costruttore di default, dato che non prende nessun parametro.

Normalmente il costruttore è definito pubblico, in modo che possa essere utilizzato per creare istanze di oggetti come ad esempio

```
MyClass mc = new MyClass();
```

Non è sempre detto che sia così, dato che utilizzando particolari pattern di programmazione si possono definire costruttori privati e delegare il compito di creare le istanze ad apposite classi che per questo motivo a volte sono denominate factory (si veda [javapattern]). Ad esempio si potrebbe avere

```
MyClass Mc = MyClassFactory.getMyClass();
```

In Java ogni oggetto definito deve obbligatoriamente avere un costruttore; nel caso non lo si definisca esplicitamente, il compilatore, al momento della compilazione, ne inserisce uno vuoto che non riceve parametri in input (per questo motivo è detto di default) e che in questo caso non compie alcuna azione.

Nel caso in cui si definisca un costruttore, allora il costruttore di default non verrà generato, e si avrà un errore in esecuzione nel tentativo di invocarlo.

In Java non è possibile definire funzioni (compresi i costruttori) con valori default, come avviene ad esempio nel linguaggio C. Per esempio, una scrittura del genere

```
public myFunction(int valore=10) {}
```

non è consentita, ma si può utilizzare una tecnica apposita, che in genere si adotta per i costruttori. Ad esempio è piuttosto frequente trovare situazioni come

```
// Costruttore di default
```

```
public MyClass() {
   this("stringa_di_default");
}

// Un altro costruttore
public MyClass(String str) {}
```

dove il costruttore di default non svolge nessuna operazione ma chiama semplicemente il costruttore con parametri passandogli un valore di default come parametro.

Si noti in questo caso l'utilizzo della keyword `this`, la quale è un reference interno alla classe stessa. Scrivendo `this()` di fatto si chiama un costruttore, ma è possibile fare riferimento a una qualsiasi proprietà o metodo della classe stessa scrivendo un'espressione del tipo

```
this.myMethod()
```

Funzione analoga alla `this` è svolta dalla parola `super` che serve per fare riferimento alla classe padre rispetto a quella in uso. Da notare che, proprio a causa della mancanza dei puntatori, da una determinata classe non è possibile fare riferimento a quelle superiori appartenenti alla stessa scala gerarchica, fatta eccezione per il padre accessibile proprio grazie alla `super`.

Da notare che, dall'interno di una classe, scrivere

```
this.myMethod()
```

oppure direttamente

```
myMethod()
```

è del tutto equivalente: il `this` in genere viene utilizzato per rendere il codice più elegante e comprensibile.

In particolare risulta essere particolarmente utile, e in questo caso non solo per motivi "estetici", quando un metodo è presente sia nella classe padre che in quella derivata, e si desidera di volta in volta specificare quale versione del metodo invocare.

Un costruttore non può restituire nessun valore, dato che implicitamente restituisce un oggetto della classe stessa. Non è proibito dichiarare un metodo che abbia il nome della classe ma che ritorni un valore, come ad esempio

```
// semplice metodo
public int MyClass() {}
```

che però viene visto come semplice metodo e non come costruttore di classe.

Oltre che attraverso l'invocazione del costruttore, in Java per creare un oggetto è possibile clonare l'area di memoria per mezzo del metodo `clone()`. Tale metodo, utilizzabile con ogni tipo di oggetto, essendo proprio della classe `Object`, effettua una copia fedele dell'area di memoria e quindi restituisce un clone identico a quello di partenza. Ad esempio

```
MyClass Mc2 = Mc1.clone();
```

Da notare che in questo caso `Mc1` e `Mc2` sono due oggetti differenti anche se identici, e che per creare `Mc2` non è stato invocato il costruttore della classe `MyClass`: ecco un altro esempio di creazione senza l'invocazione diretta del costruttore tramite una new ma grazie ad un metodo factory.

Si può dire che `Mc2` prende vita quindi nel momento in cui viene invocato il metodo clone del reference `Mc1`, e quindi nasce in funzione dello stato in cui `Mc1` si trova.

Istanziazione delle variabili di classe

Un aspetto di minore importanza, ma che risulta comunque utile in alcuni casi particolari, è relativo al modo in vengono istanziate le variabili di una classe.

Per le variabili di tipo primitivo a scope di classe (ovvero quelle definite al di fuori di ogni metodo) esse sono istanziate al loro valore di default al momento dell'istanziazione della classe, mentre per le variabili di tipo reference in genere viene assunto il valore *null*.

Il `null` è un valore speciale: di fatto non contiene nessun valore specifico, ma funge da "tappo" per evitare di avere qualcosa di molto simile ai puntatori nudi del C. In pratica, il compilatore permette di utilizzare una variabile che sia stata "tappata" con `null`, anche se poi in fase di esecuzione questo porta in genere a un errore.

Polimorfismo e programmazione dinamica

Java è un linguaggio fortemente tipato e per questo la gestione dell'interazione fra variabili (primitive e reference) differenti fra loro è di fondamentale importanza.

Nella tabella 2.1 sono illustrati tipi primitivi e reference utilizzabili; fatta eccezione per alcune particolarità, si potrà notare come vi sia una notevole somiglianza con i corrispettivi tipi messi a disposizione dagli altri linguaggi.

In Java i tipi disponibili sono quelli che si è abituati a trovare nei più importanti linguaggi procedurali e a oggetti.

Per quanto riguarda i tipi primitivi, la regola fondamentale da seguire è piuttosto semplice e si basa sulla precisione dei tipi: è consentita infatti e senza nessuna forzatura espli-

Tabella 2.1 – *I tipi di variabli presenti in Java.*

<table>
<tr><th colspan="3">tipi</th><th>keyword</th><th>note</th></tr>
<tr><td rowspan="8">primitivi</td><td colspan="2">booleani</td><td><code>boolean</code></td><td>true, false</td></tr>
<tr><td rowspan="7">numerici</td><td rowspan="5">interi</td><td><code>byte</code></td><td>8 bit interi in compl. a 2</td></tr>
<tr><td><code>short</code></td><td>16 bit</td></tr>
<tr><td><code>int</code></td><td>32 bit</td></tr>
<tr><td><code>long</code></td><td>64 bit</td></tr>
<tr><td><code>char</code></td><td>16 bit Unicode</td></tr>
<tr><td rowspan="2">floating point</td><td><code>float</code></td><td>32 bit IEEE 754</td></tr>
<tr><td><code>double</code></td><td>64 bit</td></tr>
<tr><td rowspan="2">reference</td><td colspan="2">classi</td><td><code>class</code></td><td rowspan="3"></td></tr>
<tr><td colspan="2">interfacce</td><td><code>interface</code></td></tr>
<tr><td colspan="3">null</td><td></td></tr>
</table>

cita, la conversione da un tipo a precisione minore verso uno a precisione maggiore. Ad esempio, supponendo di avere

```
int i = 10;
long j = 10;
```

è possibile scrivere senza incorrere in errori

```
j = i;
```

mentre per effettuare la conversione opposta si deve forzare l'assegnazione per mezzo di una operazione di cast

```
i = (int)j;
```

Ovviamente in questo caso, se in `j` è memorizzato un valore che supera la dimensione massima supportata da `i`, allora si avranno risultati imprevedibili, dovuti al taglio dei bit in eccesso.

Per quanto riguarda i reference si segue essenzialmente lo stesso principio, tenendo conto che l'assegnazione, sia automatica che forzata, può avvenire solo fra tipi appartenenti alla stessa scala gerarchica. In questo caso infatti la precisione del numero di bit che una variabile può contenere non è di particolare rilievo. Supponendo che sia

```
public class A {
   int a1;
   int a2;
}
```

```
public class B extends A{
   int b1;
   int b2;
}
```

allora, in base al principio della ereditarietà, B implementerà tutte le caratteristiche di A (quindi possiede le variabili `a1` e `a2`), aggiungendone eventualmente delle altre (`b1` e `b2`).

La classe B quindi possiede 4 variabili di classe, mentre la A solamente 2. Si tralasciano le considerazioni sui metodi, essendo del tutto analoghe.

In tal senso il contenuto di una classe può essere visto come una estensione del concetto di precisione dei tipi primitivi.

La conversione quindi può avvenire in maniera diretta da B ad A semplicemente eliminando le proprietà in eccesso; il contrario non è altrettanto automatico, dato che per passare da A a B non è possibile sapere come valorizzare quelle variabili presenti solo nella classe figlio.

Quindi si può scrivere

```
A a = new A();
B b = new B();
```

allora in base alla conversione implicita sarà possibile scrivere

```
a = b;
```

Questo tipo di cast viene detto implicito o upcast, dato che si risale nella gerarchia delle classi.

Il downcast invece deve essere forzato e quindi si può scrivere

```
b =(B)a;
```

Si faccia attenzione comunque che, pur non ricevendo nessun errore in fase di compilazione, tentando di eseguire questa ultima riga, si riceve un messaggio di errore del tipo `RuntimeCastException`: il downcast è possibile infatti solo nel caso in cui `b` discenda da `a`, ovvero proprio l'opposto del caso in esame.

Per chiarire meglio questo punto, che spesso infatti genera non pochi dubbi, si pensi ad esempio alla classe `Vector` e a come gestisce gli oggetti contenuti al suo interno. Questo container è in grado di contenere ogni tipo di classe Java, dato che lavora con oggetti istanze di `Object`, dalla quale classe tutti gli oggetti Java derivano.

Quindi ogni volta che si esegue una operazione del tipo

```
MyObject MObj = new MyObject();
Vector V = new Vector();
V.addElement(MObj);
```

la variabile `MyObject` viene "castata" (si tratta di un upcast) a tipo `Object` e memorizza-

ta all'interno di `Vector`. Nel momento in cui si vuole prelevare l'oggetto memorizzato nella posizione `i`, si deve scrivere

```
MyObject MObj = (MyObject)v.elementAt(i);
```

In questo caso si è effettuato un downcast, possibile proprio perché la classe `MyObject` deriva da `Object`.

Rimanendo invece nel caso in esame delle classi A e B, si sarebbe dovuto scrivere

```
A a = new B();
b = (B)a;
```

I meccanismi di upcast e downcast sono alla base di uno dei concetti più importanti in Java e nella programmazione a oggetti in genere, ovvero il polimorfismo.

Ecco brevemente di cosa si tratta. Si supponga di avere

```
public class A {
   String class_type;

   // costruttore
   public A() {
      class_type = "A";
   }

   public void metodo() {
      System.out.println("Messaggio dalla classe A");
   }

}


public class  B  extends A {

   String class_type;

   // costruttore
   public B() {
      class_type = "B";
   }

   public void metodo() {
      System.out.println("Messaggio dalla classe B");
   }

}
```

In questo caso si hanno due classi, A e B: la seconda ridefinisce sia la proprietà che il metodo, in modo da identificare su quale tipo sono invocati. Infatti ad esempio scrivendo

```
A a = new A();
a.metodo();
```

si ottiene come risultato il seguente messaggio

```
C:\> Messaggio dalla classe A
```

mentre analogamente

```
B b = new B();
b.metodo();
```

si ottiene

```
C:\> Messaggio dalla classe B
```

Comportamento del tutto analogo se si fa riferimento alle variabili di classe, ovvero eseguendo le due righe di codice seguenti

```
System.out.println("Esecuzione della classe " + a.class_type);
System.out.println("Esecuzione della classe " + b.class_type);
```

si ottengono i seguenti messaggi

```
Esecuzione della classe A
Esecuzione della classe B
```

In questo caso, facendo riferimento a un reference di B, si eseguono i metodi di B. Se invece, utilizzando l'upcast, si scrive

```
a = b;
a.metodo();
```

il risultato sarà

```
C:\> Messaggio dalla classe B
```

e non

```
C:\> Messaggio dalla classe A
```

ovvero il metodo effettivamente eseguito dipende non dal tipo del reference ma dal contenuto effettivo. Questo meccanismo prende anche il nome di binding dinamico, sebbene tale definizione venga utilizzata raramente essendo culturalmente legata alla programmazione dinamica del C++.

Comportamento opposto si ha nel caso delle variabili di classe: infatti in questo caso la variabile utilizzata è quella del tipo del reference, non del contenuto. Quindi scrivendo

```
a = b;
System.out.println("Esecuzione della classe " + a.class_type);
```

il risultato ottenuto sarà

```
C:\> Esecuzione della classe A
```

Operatori in Java

La maggior parte della sintassi di Java è simile a quella del C++ e questo vale anche per gli operatori. In Java sono disponibili gli operatori riportati nelle tabelle seguenti.

Tabella 2.2 – *Operatori aritmetici.*

simbolo	significato	note
+	Addizione aritmetica o concatenazione di stringhe	È l'unico operatore ridefinito
-	Sottrazione aritmetica	
*	Moltiplicazione aritmetica	
\	Divisione aritmetica	
=	Assegnazione	Può essere combinato con gli altri operatori aritmetici come ad esempio +=, *=

Tabella 2.3 – *Operatori booleani e di confronto.*

simbolo	significato	note
&&	AND logico	
||	OR logico	
!	Negazione	
==	Confronto	Confronta le locazioni di memoria, non il loro contenuto.

Tabella 2.4 – *Operatori bit a bit.*

simbolo	significato	note
&	AND bit a bit	
|	OR inclusivo bit a bit	
^	OR esclusivo bit a bit	
>>	Shift a destra	Le posizioni sul lato sinistro vengono riempite con bit coincidenti con il bit più significativo (bit di segno)
>>>	Shift a destra	Le posizioni sul lato sinistro vengono riempite con zeri
<<	Shift a sinistra	Le posizioni sul lato destro vengono riempite con zeri.

Tabella 2.5 – *Altri operatori.*

simbolo	significato	note
`?:`	operatore condizionale	Può essere utilizzato per scrivere un costrutto `if-then-else` in modo compatto.

Il significato di tali operatori dovrebbe apparire piuttosto ovvio, ma forse è bene fare alcune osservazioni: per prima cosa, è fondamentale tener presente che in Java non è possibile ridefinire a proprio piacimento il significato e comportamento degli operatori, come invece avviene in C/C++. L'unico operatore che ha subito un processo di overloading è il segno "+" che può essere utilizzato come somma algebrica, ma anche come concatenatore di stringhe. Ad esempio

```
String str = "salve " + "mondo " + "!"
```

produce una stringa "salve mondo !".

L'utilizzo del + come concatenatore di stringhe permette alcuni trucchi per ridurre il numero di righe di codice. Ad esempio in forma compatta è possibile scrivere

```
String str = "" + 3;
```

dove in questo caso la stringa vuota ("") viene utilizzata per forzare la conversione da intero a stringa. Si faccia attenzione che non si tratta di un qualche tipo di casting, ma piuttosto di una forma contratta che viene sostituita automaticamente dal compilatore con le seguenti

```
String str;
Integer Int = new Integer (3);
String temp = Int.toString();
str = temp;
```

In questo caso si è utilizzato il metodo `toString` ridefinito nella classe wrapper `Integer`. Il metodo `toString` è presente in tutti gli oggetti Java, dato che è definito nella classe `Object`: dove non ridefinito esplicitamente al fine di ottenere un risultato particolare, in genere restituisce una stringa rappresentazione dell'oggetto e della sua posizione nella area di memoria.

Gestione delle eccezioni

Il concetto di eccezione, pur non essendo una prerogativa esclusiva di Java, ha avuto un

nuovo impulso proprio grazie a questo linguaggio che ne fa uno dei suoi punti di forza. Si può addirittura dire che, senza il meccanismo di gestione delle eccezioni, buona parte della struttura di Java non avrebbe ragione di esistere.

Si potrebbe definire sinteticamente il concetto di eccezione come un'anomalia di funzionamento del programma che non può essere prevista in fase di scrittura del codice.

Si può prendere ad esempio il caso in cui la connessione verso un database remoto venga chiusa per un qualche motivo, oppure il collegamento via socket verso una applicazione server-side non sia più disponibile per motivi non prevedibili o non dipendenti dalla correttezza del codice dell'applicazione.

Spesso fonte di problemi è data dagli effetti derivanti dalla contestualizzazione delle variabili (effettuare una divisione potrebbe essere impossibile se il divisore è nullo), ed anche la presenza di situazioni dinamiche, polimorfiche può portare alla non predicibilità del flusso delle operazioni.

Una eccezione quindi non è un errore di programmazione, situazione questa ben più grave, ma un evento non previsto e non dipendente dalla volontà del programmatore.

Cosa sono le eccezioni e come si gestiscono

La leggenda narra che i progettisti che all'epoca definirono tutta la struttura del linguaggio fossero appassionati di uno degli sport più popolari negli USA, il baseball, ed è per tale ragione che nell'ambito della gestione delle eccezioni si trovano termini tipici di tale sport, come lanciare (`throw`) o catturare (`catch`) una eccezione.

La struttura sintattica di base è un qualcosa del tipo

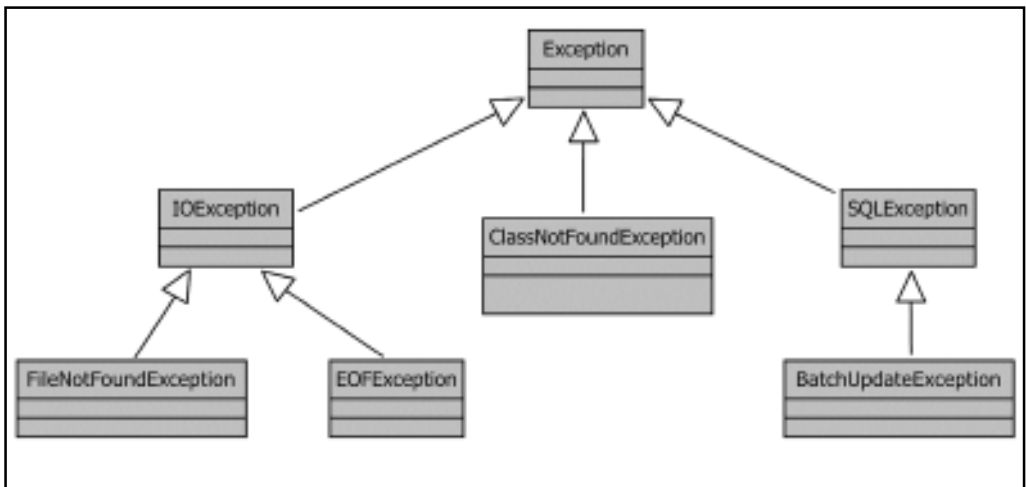
```
try {
   ...
   // porzione di codice che potrebbe dare dei problemi
   ...
}
catch(Exception1 e) {
   ...
   // gestione dell'eventuale problema nato nel blocco try
   ...
}
catch(Exception2 e) {
   ...
   // gestione dell'eventuale problema nato nel blocco try
   ...
}

finally {
   ...
   // codice da eseguire sempre
}
```

Questa organizzazione del codice permette di eseguire quella porzione di codice in cui si potrebbero verificare delle situazioni incongruenti in una zona protetta (all'interno del blocco `try`) e di gestire le eccezioni che si potrebbero verificare (blocco `catch`).

Grazie a tale organizzazione, in concomitanza del verificarsi di una eccezione, evento che normalmente può portare al blocco irrimediabile della applicazione, il controllo viene passato al costrutto `catch` dove si possono implementare soluzioni di gestione di tale anomalia (dal settare le variabili con valori opportuni a un semplice messaggio di avvertimento per l'utente).

Il blocco `catch` potrebbe essere anche vuoto, ma è necessario che sia presente. Infine si può pensare di inserire il blocco `finally` per eseguire tutte quelle istruzioni che debbano essere effettuate comunque. Si tenga presente che la `catch` è in grado di gestire solo il tipo di eccezione per la quale è definita (ovvero quella del tipo specificato fra parentesi). Quando una `catch` corrisponde all'eccezione verificatasi, il controllo passa alla parte di codice in essa contenuta e non prosegue nelle seguenti, passando eventualmente al blocco `finally`.

Il grosso vantaggio derivante dalla organizzazione ad Object Oriented di Java, è che anche le eccezioni sono oggetti, e che sono organizzate in maniera gerarchica, come mostrato nella fig. 2.5.

Figura 2.5 – *Esempio di gerarchia di alcune delle eccezioni più frequenti.*

Tale organizzazione gerarchica rispecchia la genericità: così la `Exception` essendo la più generica possibile, permette di intrappolare ogni tipo di eccezione verificatasi, mentre la `IOException` è più specifica dato che si trova più in basso nella scala gerarchica.

Quando si genererà un errore di I/O quindi verrà prodotta dal runtime una `IOException`, e tale eccezione potrà essere intrappolata con una istruzione del tipo

```
catch(IOException ioe)
```

ma anche con una

```
catch(Exception e)
```

dato che una `IOException` è anche una `Exception`. Ovviamente non è vero il contrario.

Questo tipo di organizzazione permette di dar vita a un controllo a grana più o meno fine in funzione delle proprie esigenze. Da un punto di vista implementativo questo si traduce nel posizionare i vari `catch` in ordine crescente di genericità.

Ad esempio, nella porzione di codice che segue, si ha la possibilità di far eseguire il blocco `catch` voluto in funzione del tipo di eccezione

```
try {}
catch(NullPointerException npe) {}
catch(FileNotFoundException fnfe) {}
catch(IOExcetpion ioe) {}
catch(Exception e) {}
```

In questo caso il controllo su `FileNotFoundException` deve precedere la `IOException`, dato che la seconda è più generica e potrebbe verificarsi anche in concomitanza di altre tipologie di eccezioni I/O, non solo per mancanza di un file.

`NullPointerException` (che si può verificare ad esempio se si tenta di accedere a un elemento nullo contenuto in un vettore) invece non ha niente a che vedere con le `IOException`, per cui la `catch` relativa può essere posizionata indifferentemente prima o dopo le `catch` di I/O; sicuramente deve essere posizionata prima dell'ultima `catch`.

La scelta dell'ordine delle `catch` deve essere fatto in base alla genericità controllando la gerarchia delle classi: in ogni caso il compilatore, in presenza di un ordine errato, interrompe la compilazione producendo un segnale di errore del tipo "`code not reachable`".

Propagazione delle eccezioni

Si è appena visto come sia possibile controllare l'eventuale insorgere di problemi direttamente nel luogo della loro nascita: senza interrompere l'esecuzione del programma, con una `try-catch` si può ovviare localmente a tale problema.

Non è detto però che questo modo di operare sia il migliore né quello desiderato. Si pensi ad esempio al caso in cui una certa classe funzioni come interfaccia verso un database. In tal caso, al verificarsi di una `SQLException`, l'azione necessaria è impedire che tutta l'applicazione che utilizza tale classe si blocchi o termini. Ad esempio, se la classe si chiama `DBEngine` ed il metodo in questione `executeSQL`, allora si potrebbe avere

```
public class DBEngine {
   ...
   altri metodi della classe
   ...
   public void executeSQL (String sqlstm) {
      try {}
      catch(SQLException sqle) {
      }
   } // fine metodo
} // fine classe
```

Benché formalmente corretto, questo modo di gestire l'eccezione impedisce al resto dell'applicazione di essere informata sull'insorgere dei vari problemi. Nasce quindi l'esigenza di implementare un qualche meccanismo che consenta sia di gestire l'eccezione sia di avvertire le altre parti dell'applicazione dell'evento verificatosi.

Questo potrebbe permettere ad esempio di propagare tutti gli errori, verificatisi nelle varie parti del programma, verso un unico punto deputato alla produzione dei messaggi di log, o alla comunicazione dell'utente dei problemi verificatisi. Java risolve questo problema in maniera semplice ed elegante.

Si supponga di avere un metodo `myMethod` all'interno del quale siano effettuate alcune operazioni potenzialmente pericolose. In questo caso invece di utilizzare il costrutto `try-catch` che intrappola le eccezioni localmente all'interno del metodo, si può pensare di propagare l'eccezione all'esterno del metodo per una gestione locale nella parte di codice che ha invocato il metodo stesso.

Per fare ciò è necessario e sufficiente eliminare ogni blocco `try-catch` all'interno del metodo e riscriverne la firma utilizzando la parola chiave `throws` nel seguente modo

```
public void myMethod (String str) throws Exception {
    …
}
```

In questo modo il metodo dichiara che potenzialmente al suo interno potranno verificarsi operazioni pericolose e che quindi la sua invocazione dovrà essere protetta tramite un apposito blocco `try-catch`.

Ad esempio

```
try {
   myMethod ("salve mondo");
}
catch(Exception sqle) {
   ...
   fai qualcosa con l'eccezione verificatasi
}
```

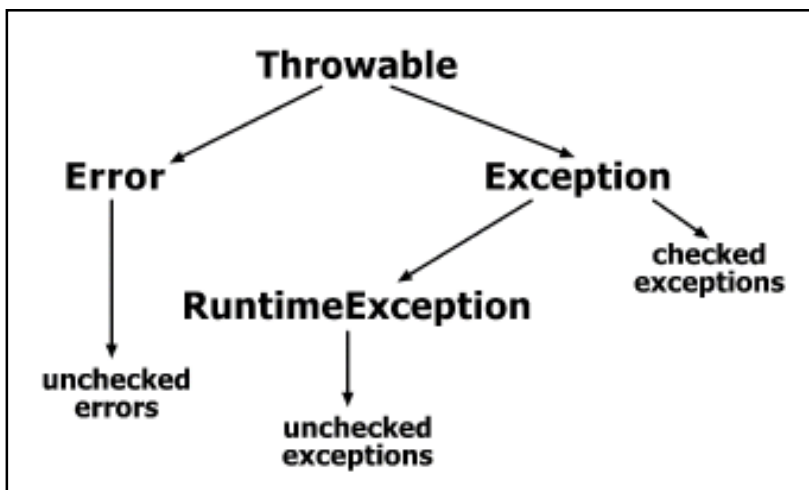
Al solito nella clausola `catch` si deve necessariamente catturare un'eccezione del tipo dichiarato nella `throws` del metodo, o di un tipo che stia più in alto nella scala gerarchica.

È sfruttando questo meccanismo che il compilatore riconosce quando un determinato metodo deve essere protetto con una `try-catch` e non tramite un conoscenza a priori sulla natura del metodo e delle operazioni eseguite all'interno.

Tipi di eccezioni

In Java si hanno due tipi di eccezioni, le checked e le unchecked: la differenza principale è che le seconde non devono essere obbligatoriamente controllate, cosa che invece è necessaria per le eccezioni di tipo checked.

Nella figura 2.6 seguente è riportato lo schema gerarchico degli errori in Java

Figura 2.6 – *Distinzione fra errori ed eccezioni, checked e unchecked.*

Come si può vedere sia la classe `Error` che la `Exception` derivano dalla più generica `Throwable`, la quale ha lo scopo di permettere a un qualsiasi oggetto che la dichiari come classe padre di generare eccezioni.

In ogni caso tutto quello che sta sotto `Error` è di fatto unchecked, e quindi non obbligatorio da controllare.

Dato che una eccezione non gestita può portare nei casi peggiori a crash dell'applicazione, l'insorgere di un errore piuttosto che di una eccezione deve essere considerata una situazione più grave.

Sempre osservando la figura 2.6 si può comprendere che anche le `RuntimeException` sono eccezioni unchecked: la loro generazione in fase di esecuzione provoca per questo un blocco dell'applicazione.

Appendice: programmare con le interfacce

Una delle linee guida fondamentali nella progettazione OO è la separazione fra l'interfaccia di una classe e la sua implementazione. A tal proposito i progettisti di Java hanno dotato il linguaggio di un costrutto, l'interfaccia appunto, distinto da quello di classe.

In questo modo si hanno a livello di implementazione due strumenti distinti per definire gli aspetti comportamentali (interfaccia) e quelli implementativi (classe).

Si consideri ad esempio il caso dell'interfaccia `Runnable`:

```
public interface Runnable {
   public abstract void run();
}
```

Una classe che voglia implementare l'interfaccia `Runnable` dovrà definire un metodo `run()` che conterrà il codice da eseguire in modalità threaded.

Una classe client che debba mandare in esecuzione un thread conterrà codice simile al seguente

```
public class Test {
   public void runIt(Runnable obj) {
       ...
       obj.run();
       ...
   }
}
```

Il principale vantaggio derivante dall'utilizzo dell'interfaccia `Runnable` nel codice precedente consiste nel fatto che il metodo `runIt()` accetta come parametro oggetti di classe diversa, senza alcun legame fra di loro se non il fatto che tutti devono implementare l'interfaccia `Runnable`. Il metodo `runIt()` è quindi ampiamente riutilizzabile e viene garantita inoltre anche la massima estendibilità: non si ha infatti alcun vincolo sulle classi che implementano l'interfaccia e nulla vieta, ad esempio, di includere funzionalità avanzate come il pooling di thread.

Il legame fra una classe che implementa un'interfaccia e i suoi client è rappresentato dai parametri dei suoi metodi. Per avere il massimo disaccoppiamento occorre quindi fare in modo che i parametri delle interfacce siano tipi predefiniti oppure interfacce, ma non classi concrete.

L'introduzione delle interfacce nel design permette quindi di ridurre le dipendenze da classi concrete ed è alla base di uno dei principi fondamentali della programmazione a oggetti: "Program to an interface, not an implementation".

È importante a questo punto osservare come un'interfaccia rappresenti un contratto fra la classe che la implementa e i suoi client. I termini del contratto sono i metodi dichiarati nell'interfaccia, metodi che ogni classe che implementi l'interfaccia si impegna a definire. Sul comportamento dei metodi però non è possibile specificare nulla in Java e altri linguaggi come C++ se non attraverso la documentazione.

Interfacce e polimorfismo

Il polimorfismo ottenuto attraverso l'ereditarietà è uno strumento molto potente. Le interfacce ci permettono di sfruttare il polimorfismo anche senza ricorrere a gerarchie di ereditarietà. Ecco un esempio

```
public interface PricedItem {
   public void setPrice(double price);
   public double getPrice();
}
```

L'interfaccia `PricedItem` definisce il comportamento di un articolo con prezzo. Possiamo a questo punto implementare l'interfaccia in una classe applicativa `Book` nel modo seguente

```
public class Book implements PricedItem {
   private String title;
   private String author;
   private double price;
   ...
   public void setPrice(double price) {
      this.price = price;
   }
   public double getPrice() {
      return price;
   }
}
```

Il codice client che necessita del prezzo di un libro può quindi essere scritto così:

```
double computeTotalPrice(Collection items) {
   Iterator it = items.iterator();
   PricedItem item;
      double total = 0;
      while (it.hasNext()) {
         item = (PricedItem)it.next();
```

```
         total += item.getPrice();
      }
   return total;
}
```

Il metodo precedente calcola il prezzo totale di una collezione di oggetti. Si supponga ora di voler estendere l'applicazione per gestire non solo libri ma anche CD musicali. Si può pensare di introdurre a questo proposito una nuova classe che implementa l'interfaccia `PricedItem`

```
public class CD implements PricedItem {
   private String title;
   private String singer;
   private Collection songs;
   private double price;
   ...
   public void setPrice(double price) {
      this.price = price;
   }
   public double getPrice() {
      return price;
   }
}
```

Il codice precedente per il calcolo del prezzo totale funziona senza modifiche anche se la collezione contiene oggetti di classe CD perché tale codice si riferisce all'interfaccia e non alla classe concreta.

Ereditarietà multipla

Come noto Java non supporta l'ereditarietà multipla fra classi. Una classe può però implementare più interfacce e in questo modo è possibile definire per essa diversi comportamenti. Si riconsideri ora l'esempio precedente aggiungendo il supporto alla persistenza.

L'interfaccia `Persistent` è ciò che serve per questo scopo

```
public interface Persistent {
   public void save();
}
```

Le classi viste in precedenza diventano quindi

```
public class Book implements PricedItem, Persistent {
```

```
   ...
}

public class CD implements PricedItem, Persistent {
   ...
}
```

Quindi si può scrivere il codice di gestione del salvataggio nel seguente modo

```
public void saveAll(Collection items) {
   Iterator it = items.iterator();
   Persistent item;
   while (it.hasNext()) {
      item = (Persistent)it.next();
      item.save();
   }
}
```

Si osservi che l'interfaccia `Persistent` nasconde completamente i dettagli di salvataggio che potrebbe avvenire su file oppure su DB attraverso JDBC.

Composizione

La programmazione OO permette di riutilizzare funzionalità esistenti principalmente attraverso ereditarietà fra classi e composizione di oggetti. La composizione permette di ottenere sistemi più flessibili mentre l'ereditarietà dovrebbe essere utilizzata principalmente per modellare relazioni costanti nel tempo. Non si pensi però di poter ottenere il polimorfismo solo con l'ereditarietà: l'utilizzo combinato di interfacce e composizione permette di progettare soluzioni molto interessanti dal punto di vista architetturale.

Si supponga infatti di dover sviluppare il supporto alla validazione per le classi `Book` viste prima. Le logiche di validazione saranno incorporate all'interno di un'opportuna classe che implementa un'interfaccia `Validator`

```
public interface Validator {
   public void validate(Object o);
   }

public class BookValidator implements Validator {
   public void validate(Object o) {
      if (o instanceof Book) {
      ...
      }
   }
}
```

Ecco la classe che si occupa di eseguire la validazione

```
public class Manager {
   ...
   Validator validator;
   public Manager(Validator validator) {
      this.validator = validator;
      ...
   }
   public void validate() {
      ...
      validator.validate(object)
      ...
   }
}
```

La classe `Manager` non fa riferimento alla classe concreta `BookValidator`, quindi è possibile cambiare la logica di validazione anche a run-time. La soluzione di design che è stata illustrata è nota come Pattern Strategy.

Interfacce che estendono altre interfacce

Come le classi anche le interfacce possono essere organizzate in gerarchie di ereditarietà. Ad esempio

```
interface Base {
   ...
}
interface Extended extends Base {
   ...
}
```

L'interfaccia `Extended` eredita quindi tutte le costanti e tutti i metodi dichiarati in `Base`. Ogni classe che implementa `Extended` dovrà quindi fornire una definizione anche per i metodi dichiarati in `Base`. Le interfacce possono poi derivare da più interfacce, allo stesso modo in cui una classe può implementare più interfacce.

```
interface Sibling { ... }
interface Multiple extends Base, Sibling { ... }
```

Si vedrà ora come vengono gestiti i conflitti di nomi. Se due interfacce contengono due metodi con la stessa signature e con lo stesso valore di ritorno, la classe concreta dovrà implementare il metodo solo una volta e il compilatore non segnalerà alcun errore. Se i

metodi hanno invece lo stesso nome ma signature diverse allora la classe concreta dovrà dare un'implementazione per entrambi i metodi. I problemi si verificano quando le interfacce dichiarano due metodi con gli stessi parametri ma diverso valore di ritorno. Per esempio

```
interface Int1 {
   int foo();
}
interface Int2 {
   String foo();
}
```

In questo caso il compilatore segnala un errore perché il linguaggio non permette che una classe abbia due metodi la cui signature differisce solo per il tipo del valore di ritorno. Consideriamo infine il caso in cui due interfacce dichiarino due costanti con lo stesso nome, eventualmente anche con tipo diverso. La classe concreta potrà utilizzare entrambe le costanti qualificandole con il nome dell'interfaccia.

Interfacce e creazione di oggetti

Come è stato visto le interfacce permettono di astrarre dai dettagli implementativi, eliminare le dipendenze da classi concrete e porre l'attenzione sul ruolo degli oggetti nell'architettura che si vuole sviluppare. Rimane però un problema relativo alla creazione degli oggetti: in tale situazione occorre comunque specificare il nome di una classe concreta. In questo caso si genera quindi una dipendenza di creazione.

Ad esempio

```
public class MyDocument {
   ...
   public void open();
   public void close();
   public void save();
   ...
}
MyDocument doc = new MyDocument();
```

Nell'istruzione precedente si crea un oggetto di classe concreta `MyDocument` ma il codice non potrà essere riutilizzato per creare un oggetto di classe estesa da `MyDocument` oppure un'altra classe che rappresenta un diverso tipo di documento. È possibile risolvere questo problema creando classi final oppure rendendo ridefinibile l'operazione di creazione. Quest'ultima soluzione è senz'altro la più interessante e i pattern creazionali ci permettono di risolvere il problema. Si vedrà ora come applicare il pattern Abstract Factory

per incapsulare il processo di creazione ed eliminare la dipendenza di creazione di cui soffriva il precedente esempio. Innanzi tutto si introduce un'interfaccia per rappresentare un documento

```
public interface Document {
   public void open();
   public void close();
   public void save();
}
```

A questo punto si definisce un'interfaccia per un factory, cioè un oggetto il cui compito è quello di creare altri oggetti. Poiché a priori non si sa quale tipo di oggetti dovranno essere creati, si ricorre alle interfacce

```
public interface DocumentFactory {
   public Document createDocument();
}
```

È ora possibile definire diversi tipi di documento, ad esempio

```
public class TechnicalDocument implements Document {
   public void open() { ... }
   public void close() { ... }
   public void save() { ... }
}
public class CommercialDocument implements Document {
   public void open() { ... }
   public void close() { ... }
   public void save() { ... }
}
```

Ora si desidera essere in grado di creare diversi tipi di documento. Per questo si definisce una classe factory per ogni diversa classe documento

```
public class TechnicalDocumentFactory implements DocumentFactory {
   public Document createDocument() {
      Document doc = new TechnicalDocument();
      ...
      return doc;
   }
}
public class CommercialDocumentFactory implements DocumentFactory {
   public Document createDocument() {
      Document doc = new CommercialDocument();
      ...
```

```
        return doc;
    }
}
```

È possibile quindi creare oggetti documento nel modo seguente

```
void manageDocument(DocumentFactory factory) {
    Document doc = factory.createDocument();
    doc.open();
    ... // modify document
    doc.save();
    doc.close();
}
```

Il codice precedente crea un oggetto che implementa l'interfaccia `Document` ma non ha alcun legame con classi concrete e si può quindi utilizzare con classi diverse, purché conformi all'interfaccia `Document`.

Vantaggi delle interfacce nello sviluppo del software

Dopo aver passato in rassegna diversi esempi sull'utilizzo delle interfacce è possibile a questo punto discutere sui loro reali vantaggi:

le interfacce permettono innanzitutto di concentrarsi sugli aspetti comportamentali degli oggetti e costruire quindi astrazioni efficaci per il problema in esame, nascondendo i dettagli implementativi all'interno delle classi concrete;

ragionare per interfacce permette di separare le politiche di interazione fra classi dalle caratteristiche interne di una classe;

le interfacce rappresentano inoltre uno strumento per il disaccoppiamento fra classi concrete, ovvero per l'eliminazione delle dipendenze che si è visto essere deleterie per un buon design.

Rimane da approfondire un aspetto importante, ovvero come specificare condizioni più precise nell'invocazione dei metodi di un'interfaccia. L'idea è quella di vedere un'interfaccia come un contratto stipulato fra la classe che la implementa e i suoi client; il problema è quindi quello di stabilire i termini del contratto in maniera precisa e non ambigua. Esiste una metodologia di programmazione, il Design By Contract, che permette di specificare per ogni metodo le condizioni che il client deve rispettare (precondizioni), quelle garantite dal metodo (postcondizioni) e quelle sempre valide (invarianti).

Capitolo 3

Programmazione concorrente e gestione del multithreading in Java

DI PAOLO AIELLO – GIOVANNI PULITI

Introduzione

Una delle potenti caratteristiche del linguaggio Java è il supporto per la *programmazione concorrente* o *parallela*. Tale *feature* permette di organizzare il codice di una stessa applicazione in modo che possano essere mandate in esecuzione contemporanea più parti di codice differenti fra loro.

Prima di descrivere questi aspetti del linguaggio saranno introdotti alcuni concetti fondamentali che aiuteranno ad avere un'idea più chiara dell'argomento e delle problematiche correlate.

Processi e multitasking

Tutti i moderni sistemi operativi offrono il supporto per il *multitasking*, ossia permettono l'esecuzione simultanea di più *processi*. In un sistema Windows, Unix o Linux si può, ad esempio, scrivere una e-mail o un documento di testo mentre si effettua il download di un file da Internet. In apparenza questi diversi programmi vengono eseguiti contemporaneamente, anche se il computer è dotato di un solo processore.

In realtà i processori dei calcolatori su cui si è abituati a lavorare analizzano il flusso delle istruzioni in maniera sequenziale in modo che in ogni istante una sola istruzione sia presa in esame ed eseguita (questo almeno in linea di massima, dato che esistono architetture particolari che permettono il parallelismo a livello di microistruzioni).

Ma anche se, per sua natura, un computer è una macchina sequenziale, grazie a una gestione ciclica delle risorse condivise (prima fra tutte il processore centrale), si ottiene una specie di parallelismo, che permette di simulare l'esecuzione contemporanea di più programmi nello stesso momento.

Grazie alla elevata ottimizzazione degli algoritmi di gestione di questo pseudoparallelismo, e grazie alla possibilità di un processo di effettuare certi compiti quando gli altri sono in pausa o non sprecano tempo di processore, si ha in effetti una simulazione del parallelismo fra processi, anche se le risorse condivise sono in numero limitato.

Nel caso in cui si abbiano diversi processori operanti in parallelo, è possibile che il parallelismo sia reale, nel senso che un processore potrebbe eseguire un processo mentre un altro processore esegue un diverso processo, senza ripartizione del tempo: in generale non è possibile però fare una simile assunzione, dato che normalmente il numero di processi in esecuzione è maggiore (o comunque può esserlo) del numero di processori fisici disponibili, per cui è sempre necessario implementare un qualche meccanismo di condivisione delle risorse.

Un processo è un flusso di esecuzione del processore corrispondente a un programma. Il concetto di processo va però distinto da quello di programma in esecuzione, perché è possibile che un processore esegua contemporaneamente diverse istanze dello stesso programma, ossia generi diversi processi che eseguono lo stesso programma (ad esempio diverse istanze del Notepad, con documenti diversi, in ambiente Windows).

Per multitasking si intende la caratteristica di un sistema operativo di permettere l'esecuzione contemporanea (o pseudocontemporanea, per mezzo del time-slicing) di diversi processi.

Vi sono due tipi di multitasking:

il *cooperative multitasking* la cui gestione è affidata agli stessi processi, che mantengono il controllo del processore fino a che non lo rilasciano esplicitamente. Si tratta di una tecnica abbastanza rudimentale in cui il funzionamento dipende dalla bontà dal codice del programma, quindi in sostanza dal programmatore. C'è sempre la possibilità che un programma scritto in modo inadeguato monopolizzi le risorse impedendo il reale funzionamento multitasking. Esempi di sistemi che usano questo tipo di multitasking sono Microsoft Windows 3.x e alcune versioni del MacOS;

il *preemptive multitasking* è invece gestito interamente dal sistema operativo con il sistema del *time-slicing* (detto anche *time-sharing*), assegnando ad ogni processo un intervallo di tempo predefinito, ed effettuando il cambio di contesto anche senza che il processo intervenga o ne sia a conoscenza. Il processo ha sempre la possibilità di rilasciare volonta-

riamente le risorse, ma questo non è necessario per il funzionamento del sistema. Il sistema operativo in questo caso utilizza una serie di meccanismi per il controllo e la gestione del tempo del processore, in modo da tener conto di una serie di parametri, legati al tempo trascorso e all'importanza (*priorità*) di un determinato processo.

Nonostante il fatto che i termini preemptive e time-slicing abbiano significato simile, in realtà preemptive si riferisce alla capacità di un processo di "prevalere" su un altro di minore priorità sottraendogli il processore in base a tale "diritto di priorità", mentre il time-slicing, anche se generalmente coesiste con la preemption, si riferisce unicamente alla suddivisione del tempo gestita dal sistema (e non lasciata ai processi), anche tra processi a priorità uguale. Lo scheduling usato per gestire i processi a uguale priorità è generalmente il cosiddetto round-robin scheduling, in cui un processo, dopo che ha usufruito della sua porzione di tempo, viene messo in attesa in coda fra processi con uguale priorità. Sia la preemption che il time-slicing presuppongono un intervento da parte del sistema operativo nel determinare quale processo deve essere mandato in esecuzione. Comunque possono esserci sistemi preemptive che non usano il time-slicing, ma usano ugualmente le priorità per determinare quale processo deve essere eseguito. Si tornerà su questo aspetto a proposito della gestione dei thread in Java.

Si è detto che per simulare il parallelismo fra processi differenti si effettua una spartizione del tempo trascorso in esecuzione nel processore. Il meccanismo di simulazione si basa sul cambio di contesto (*context-switch*) fra processi diversi: in ogni istante un solo processo viene messo in esecuzione, mentre gli altri restano in attesa.

Il contesto di un processo P1 è l'insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il sistema al momento in cui interrompe l'esecuzione del processo P1 per passare a un altro processo P2. Tra queste informazioni di contesto le principali sono lo *stato dei registri del processore*, e la *memoria del processo*, che a sua volta contiene il *testo* del programma, ossia la sequenza di istruzioni, i *dati* gestiti dal processo e lo *stack* (spazio di memoria per le chiamate di funzioni e le variabili locali).

Infatti, un aspetto fondamentale della gestione dei processi è il fatto che *ogni processo ha un suo spazio di memoria privato*, a cui esso soltanto può accedere. Quindi, salvo casi eccezionali (memoria condivisa) un processo non ha accesso alla memoria gestita da un altro processo.

I processi sono normalmente organizzati secondo una struttura gerarchica in cui, a partire da un primo processo iniziale creato alla partenza del sistema operativo, ogni successivo processo è "figlio" di un altro processo che lo crea e che ne diviene il "padre".

Nei sistemi preemptive vi è poi un processo particolare che gestisce tutti gli altri processi, lo *scheduler*, responsabile della corretta distribuzione del tempo della CPU tra i processi in esecuzione. A tale scopo esistono diversi algoritmi di *scheduling*, che comunque generalmente si basano sul tempo di attesa (maggiore è il tempo trascorso dall'ultima esecuzione, maggiore è la priorità del processo) e su livelli di priorità intrinseci assegnati dal sistema sulla base della natura del processo, oppure dall'utente sulla base delle sue esigenze particolari. A prescindere da questo normale avvicendamento di esecuzione, i processi possono subire delle interruzioni (*interrupt*) dovute al verificarsi di eventi particolari, originati dall'hardware come l'input di una periferica (interrupt hardware), dal software (interrupt software) oppure da errori di esecuzione che causano le cosiddette *eccezioni*. In questi casi viene effettuato un context-switch come nel normale scheduling, viene eseguito del codice specifico che gestisce l'interruzione, dopodiché si torna al processo interrotto con un altro context-switch. I processi, durante il loro ciclo di vita, assumono *stati* differenti, in conseguenza del loro funzionamento interno e dell'attività dello scheduler. Semplificando al massimo, questi sono i principali stati che un processo può assumere:

- *in esecuzione*: il processo è attualmente in esecuzione;
- *eseguibile*: il processo non è in esecuzione, ma è pronto per essere eseguito, appena la CPU si rende disponibile;
- *in attesa*: il processo è in attesa di un dato evento, come lo scadere di una frazione di tempo, la terminazione di un altro processo, l'invio di dati da un canale I/O.

I processi normalmente sono entità tra loro separate ed estranee ma, qualora risultasse opportuno, sono in grado di comunicare tra di loro utilizzando mezzi di comunicazione appositamente concepiti, genericamente identificati dalla sigla IPC (Inter Process Communication). Tra questi si possono citare la memoria condivisa (*shared-memory*), i *pipe*, i *segnali*, i *messaggi*, i *socket*. A seconda della tipologia di comunicazione tra processi, possono sorgere dei problemi derivanti dall'acceso contemporaneo, diretto o indiretto, alle medesime risorse. Per evitare che questo dia origine a errori e incongruenze, generalmente le risorse vengono acquisite da un singolo processo con un *lock*, e rilasciate una volta che l'operazione è terminata. Solo a quel punto la risorsa sarà disponibile per gli altri processi. Per gestire questo tipo di problemi di *sincronizzazione* esistono appositi meccanismi, tra cui il più conosciuto è quello dei *semafori*. Per maggiori approfondimenti legati a questi argomenti si faccia riferimento alla bibliografia.

Thread e multithreading

L'esecuzione parallela e contemporanea di più tasks (intendendo per task l'esecuzione

di un compito in particolare), risulta utile non solo nel caso di processi in esecuzione su un sistema operativo multitasking, ma anche all'interno di un singolo processo.

Si pensi, ad esempio, a un editor di testo in cui il documento corrente viene automaticamente salvato su disco ogni *n* minuti. In questo caso il programma è composto da due flussi di esecuzione indipendenti tra loro: da un lato l'editor che raccoglie i dati in input e li inserisce nel documento, dall'altra il meccanismo di salvataggio automatico che resta in attesa per la maggior parte del tempo e, a intervalli prestabiliti, esegue la sua azione.

Sulla base di simili considerazioni è nata l'esigenza di poter usare la programmazione concorrente all'interno di un singolo processo, e sono stati concepiti i *thread*, i quali in gran parte replicano il modello dei processi concorrenti, applicato però nell'ambito di una singola applicazione. Un processo quindi non è più un singolo flusso di esecuzione, ma un insieme di flussi: ogni processo contiene almeno un thread (thread principale) e può dare origine ad altri thread generati a partire dal thread principale.

Come per il multitasking, anche nel multithreading lo scheduling dei thread può essere compiuto dal processo (dal thread principale), eventualmente appoggiandosi ai servizi offerti dal sistema operativo, se questo adotta il *time-slicing*; in alternativa può essere affidato ai singoli thread, ed allora il programmatore deve fare attenzione a evitare che un singolo thread monopolizzi le risorse, rilasciandole periodicamente secondo criteri efficienti.

La differenza fondamentale tra processi e thread sta nel fatto che i thread *condividono lo stesso spazio di memoria*, se si prescinde dallo stack, ossia dai dati temporanei e locali usati dalle funzioni.

Questo porta diverse conseguenze: il cambio di contesto fra thread è molto meno pesante di quello tra processi, e quindi l'uso di thread diversi causa un dispendio di risorse inferiore rispetto a quello di processi diversi; inoltre la comunicazione fra thread è molto più semplice da gestire, dato che si ha condivisione dello stesso spazio di memoria.

D'altra parte, proprio questa condivisione rende molto più rilevanti e frequenti i problemi di sincronizzazione, come si vedrà dettagliatamente in seguito.

Un thread è un flusso di esecuzione del processore corrispondente a una sequenza di istruzioni all'interno di un processo. Analogamente ai processi, bisogna distinguere il concetto di esecuzione di una sequenza di istruzioni da quello di thread, poiché ci possono essere diverse esecuzioni parallele di uno stesso codice, che danno origine a thread diversi.

Per multithreading si intende l'esecuzione contemporanea (ovvero pseudocontemporanea, per mezzo del time-sharing) di diversi thread nell'ambito dello stesso processo. La gestione del multithreading può essere a carico del sistema operativo, se questo supporta i thread, ma può anche essere assunta dal processo stesso.

I thread e la Java Virtual Machine

Si è visto che un thread è un flusso di esecuzione nell'ambito di un processo. Nel caso di Java, ogni esecuzione della macchina virtuale dà origine a un processo, e tutto quello che viene mandato in esecuzione da una macchina virtuale (ad esempio un'applicazione o una Applet) dà origine a un thread.

La virtual machine Java è però un processo un po' particolare, in quanto funge da ambiente portabile per l'esecuzione di applicazioni su piattaforme differenti. Quindi la JVM non può fare affidamento su un supporto dei thread da parte del sistema operativo, ma deve comunque garantire un certo livello minimo di supporto, stabilito dalle specifiche ufficiali della virtual machine. Queste stabiliscono che una VM gestisca i thread secondo uno scheduling di tipo preemptive chiamato *fixed-priority scheduling*. Questo schema è basato essenzialmente sulla priorità ed è preemptive perché garantisce che, se in qualunque momento si rende eseguibile un thread con priorità maggiore di quella del thread attualmente in esecuzione, il thread a maggiore priorità prevalga sull'altro, assumendo il controllo del processore.

La garanzia che sia sempre in esecuzione il thread a priorità più alta non è assoluta. In casi particolari lo scheduler può mandare in esecuzione un thread a priorità più bassa per evitare situazioni di stallo o un consumo eccessivo di risorse. Per questo motivo è bene non fare affidamento su questo comportamento per assicurare il corretto funzionamento di un algoritmo dal fatto che un thread ad alta priorità prevalga sempre su uno a bassa priorità.

Le specifiche della VM non richiedono il time-slicing nella gestione dei thread, anche se questo è in realtà presente nei più diffusi sistemi operativi e, di conseguenza può essere utilizzato dalle VM che girano su questi sistemi. Per questo motivo, se si vuole che un'applicazione Java funzioni correttamente indipendentemente dal sistema operativo e dalla implementazione della VM, non si deve assumere la gestione del time-sharing da parte della VM, ma bisogna far sì che ogni thread rilasci spontaneamente le risorse quando opportuno. Quest'aspetto sarà analizzato nei dettagli più avanti.

Si diceva che generalmente le VM usano i servizi di sistema relativi ai thread, se presenti. Ma ciò non è tassativo. Una macchina virtuale può anche farsi interamente carico della gestione dei thread, senza far intervenire il sistema operativo, anche se questo supporta i thread. In questo caso la VM è vista dal sistema come un processo con un singolo thread, mentre i thread Java sono ignorati totalmente dal sistema stesso. Questo modello di implementazione dei thread nella VM è conosciuto come *green-thread* (*green* in questo caso è traducibile approssimativamente con *semplice*) ed è adottato da diverse implementazioni della VM, anche in sistemi (ad esempio alcune versioni di Unix) in cui

esiste un supporto nativo dei thread. Viceversa in ambiente Windows, le VM usano generalmente i servizi del sistema operativo. Analogamente ai processi, i thread assumono in ogni istante un determinato stato. Nella VM si distinguono i seguenti stati dei thread:

- *initial*: un thread si trova in questa condizione tra il momento in cui viene creato e il momento in cui comincia effettivamente a funzionare;
- *runnable*: è lo stato in cui si trova normalmente un thread dopo che ha cominciato a funzionare. Il thread in questo stato può, in qualunque momento, essere eseguito;
- *running*: il thread è attualmente in esecuzione. Questo non sempre viene considerato uno stato a sé, ma in effetti si tratta di una condizione diversa dallo stato runnable. Infatti ci possono essere diversi thread nello stato runnable in un dato istante ma, in un sistema a singola CPU, uno solo è in esecuzione, e viene chiamato *thread corrente*;
- *blocked*: il thread è in attesa di un determinato evento;
- *dead*: il thread ha terminato la sua esecuzione.

La programmazione concorrente in Java

Dopo tale panoramica su programmazione parallela e thread, si può analizzare come utilizzare i thread in Java. Gli strumenti a disposizione per la gestione dei thread sono essenzialmente due: la classe `java.lang.Thread` e l'interfaccia `java.lang.Runnable`.

Dal punto di vista del programmatore, i thread in Java sono rappresentati da oggetti che sono o istanze della classe `Thread`, o istanze di una sua sottoclasse, oppure oggetti che implementano l'interfaccia `Runnable`. D'ora in avanti si utilizzerà il termine thread sia per indicare il concetto di thread, sia per far riferimento alla classe `Thread` che a una qualsiasi classe che implementi le funzionalità di un thread.

Creazione e terminazione di un thread

Inizialmente verrà presa in esame la modalità di creazione e gestione dei thread basata sull'utilizzo della classe `Thread`, mentre in seguito sarà analizzata la soluzione alternativa basata sull'interfaccia `Runnable`. La classe `Thread` è una classe *non astratta* attraverso la quale si accede a tutte le principali funzionalità per la gestione dei thread, compresa la creazione dei thread stessi. Il codice necessario per creare un thread è il seguente:

```
Thread myThread = new Thread();
```

A meno di associarvi un oggetto `Runnable`, istanziando direttamente un oggetto della

classe `Thread` però non si ottiene nessun particolare risultato, dato che esso termina il suo funzionamento quasi subito: infatti le operazioni svolte in modalità threaded sono quelle specificate nel metodo `run()`, metodo che deve essere ridefinito dalle classi derivate.

Se si desidera quindi che il thread faccia qualcosa di utile ed interessante, si deve creare una sottoclasse di `Thread`, e ridefinire il metodo `run()`. Qui di seguito è riportato un esempio

```
public class SimpleThread extends Thread {
   String message;

   public SimpleThread(String s){
   message = s;
   }

   public void run() {
      for (int i = 0; i < 100; i++)
      System.out.println(message);
   }

   public static void main(String[] args) {
      SimpleThread st1, st2;
      st1 = new SimpleThread("Buongiorno");
      st2 = new SimpleThread("Buonasera");
      st1.start();
      st2.start();
   }
}
```

Per far partire il thread, una volta creato, si usa il metodo `start()`, il quale provoca l'esecuzione del metodo `run()`; terminato tale metodo, il thread cessa la sua attività, e le risorse impegnate per quel thread vengono rilasciate. A questo punto, se all'oggetto `Thread` è collegata a una variabile in uno scope ancora attivo, l'oggetto non viene eliminato dal garbage collector a meno che la variabile non venga impostata a `null`.

Tuttavia tale oggetto non è più utilizzabile, ed una successiva chiamata del metodo `start()`, pur non generando alcuna eccezione, non avrà alcun effetto; la regola di base dice infatti che l'oggetto `Thread` è concepito per essere usato una volta soltanto.

È quindi importante tener presente che, se si hanno uno o più riferimenti, si dovrebbe aver cura di impostare tali variabili a `null` per liberare la memoria impegnata dall'oggetto. Se invece creiamo il Thread senza alcun riferimento a una variabile, ad esempio

```
new SimpleThread("My Thread").start();
```

la virtual machine si fa carico di mantenere l'oggetto in memoria per tutta la durata di esecuzione del thread, e di renderlo disponibile per la garbage collection una volta terminata l'esecuzione.

La classe `Thread` contiene anche un metodo `stop()`, che permette di terminare l'esecuzione del thread dall'esterno. Ma questo metodo è deprecato in Java 2 per motivi di sicurezza. Infatti in questo caso l'esecuzione si interrompe senza dare la possibilità al thread di eseguire un cleanup: il thread in questo caso non ha alcun controllo sulle modalità di terminazione. Per questo motivo l'uso di `stop()` è da evitare comunque, indipendentemente dalla versione di Java che si usa.

L'interfaccia `Runnable`

L'altra possibilità che permette di creare ed eseguire thread si basa sull'utilizzo della interfaccia `Runnable` a cui si accennava in precedenza. Ecco un esempio, equivalente al precedente, ma che usa una classe `Runnable` anziché una sottoclasse di `Thread`:

```
public class SimpleRunnable implements Runnable {
   String message;

   public SimpleRunnable(String s) {
      message = s;
   }

   public void run() {
      for (int i = 0; i < 100; i++)
         System.out.println(message);
   }

   public static void main(String[] args) {
      SimpleRunnable sr1, sr2;
      sr1 = new SimpleRunnable("Buongiorno");
      sr2 = new SimpleRunnable("Buonasera");
      Thread t1 = new Thread(sr1);
      Thread t2 = new Thread(sr2);
      t1.start();
      t2.start();
   }
}
```

L'interfaccia `Runnable` contiene un solo metodo, il metodo `run()`, identico a quello già visto per la classe `Thread`. Questa non è una semplice coincidenza, dal momento che la classe `Thread`, in realtà, implementa l'interfaccia `Runnable`.

Per la precisione implementando l'interfaccia `Runnable` e il metodo `run()`, una classe *non derivata da* `Thread` può funzionare come un `Thread`, e per far questo però, deve

Figura 3.1 – *Per creare una classe che sia contemporaneamente un Thread ma anche qualcos'altro, si può optare per una ereditarietà multipla. Tale soluzione non è permessa in Java.*

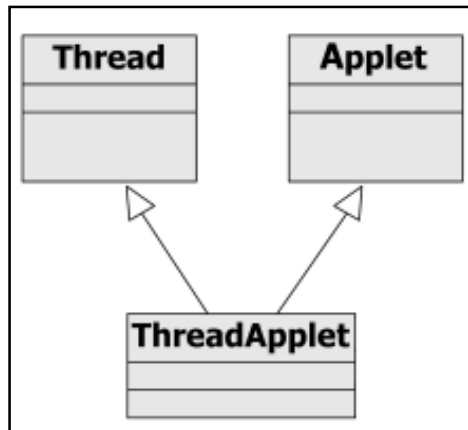



Figura 3.2 – *In alternativa, si può ereditare da un solo padre ed inglobare un oggetto che svolga la funzione che manca. Questo pattern, molto utilizzato, non risulta essere particolarmente indicato nel caso dei thread.*

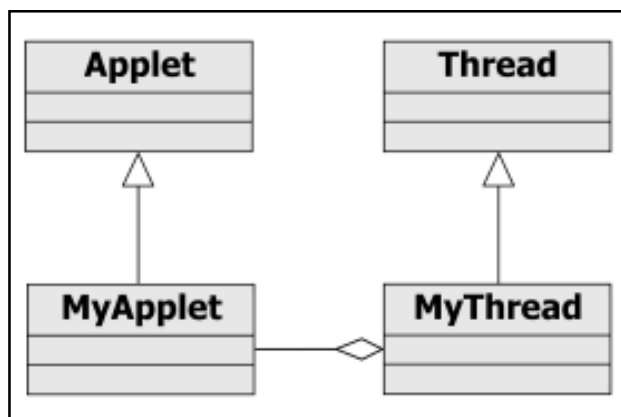
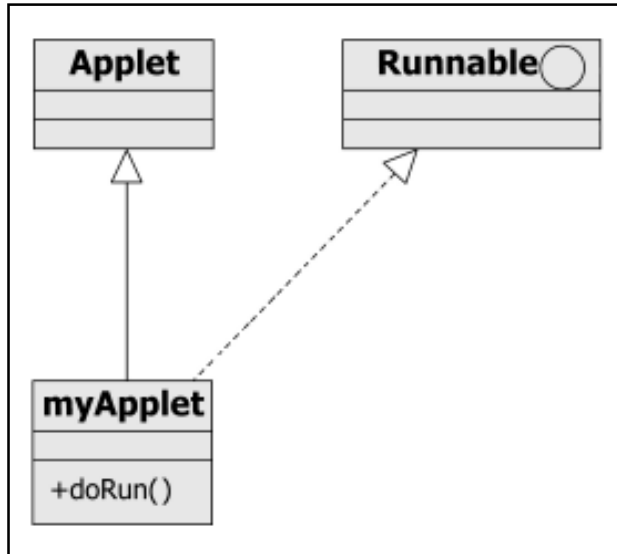



Figura 3.3 – *Grazie all'utilizzo dell'interfaccia Runnable, si possono derivare classi al fine di specializzarne il comportamento ed aggiungere funzionalità di thread.*

essere "agganciata" a un oggetto `Thread` (un'istanza della classe `Thread` o di una sua sottoclasse) passando un reference dell'oggetto `Runnable` al costruttore del `Thread`.

Dall'esempio fatto, però, l'interfaccia `Runnable` non risulta particolarmente utile, anzi sembra complicare inutilmente le cose: che bisogno c'è di rendere un altro oggetto `Runnable` se si può usare direttamente una sottoclasse di `Thread`?

Per rispondere a tale quesito, si pensi ad una situazione in cui si voglia far sì che una certa classe implementi contemporaneamente la funzione di thread, ma che specializzi anche un'altra classe base (fig. 3.1).

Ora dato che in Java non è permessa l'ereditarietà multipla, tipicamente una soluzione a cui si ricorre è quella di utilizzare uno schema progettuale differente, basato spesso sul pattern Delegation o sullo Strategy (come mostrato in fig. 3.2).

Questa architettura non si adatta molto bene al caso dei thread, o comunque risulta essere troppo complessa, visto che l'interfaccia `Runnable` ne offre un'altra molto più semplice. Derivando dalla classe base e implementando l'interfaccia `Runnable` infatti si può sia personalizzare la classe base, sia aggiungere la funzione di thread (si veda la fig. 3.3).

Ecco con un esempio come si può implementare tale soluzione

```
class RunnableApplet extends Applet implements Runnable {
   String message;
```

```
   RunnableApplet(String s) {
      message = s;
   }

   public void init() {
      Thread t = new Thread(this);
      t.start();
   }

   public void run() {
      for (int i = 0; i < 100; i++)
         System.out.println(message);
}
```

Anche se l'esempio è forse poco significativo, riesce a far capire come l'oggetto può eseguire nel metodo `run()` dei compiti suoi propri, usando i suoi dati e i suoi metodi, e anche quelli ereditati dalla classe base, mentre l'oggetto `Thread` incapsulato viene usato solo per eseguire tutto questo in un thread separato.

Utilizzando una variabile di classe per il thread possiamo incrementare il controllo sul thread: aggiungendo un metodo `start()` è possibile far partire il thread dall'esterno, al momento voluto anziché automaticamente in fase di inizializzazione dell'Applet:

```
class RunnableApplet extends Applet implements Runnable {
   String message;
   Thread thread;

   RunnableApplet(String s) {
      message = s;
   }

   public void init() {
      thread = new Thread(this);
   }

   public void start() {
      t.start();
   }

   public void run() {
      for (int i = 0; i < 100; i++)
         System.out.println(message);
}
```

Questo è il caso più tipico di utilizzo dell'interfaccia `Runnable`: un oggetto `Thread` viene inglobato in un oggetto già derivato da un'altra classe e utilizzato come "motore" per l'esecuzione di un certo codice in un thread separato.

Quindi si può dire semplicisticamente che l'uso dell'interfaccia `Runnable` al posto della derivazione da `Thread` si rende necessario quando la classe che si vuole rendere `Runnable` è già una classe derivata.

Negli altri casi si può scegliere il metodo che appare più conveniente.

Identificazione del thread

Ogni thread che viene creato assume un'identità autonoma all'interno del sistema: per facilitarne la successiva identificazione è possibile assegnare un nome al thread, passandolo al costruttore. Ad esempio:

```
SimpleRunnable sr = new SimpleRunnable("Buongiorno");
Thread t = new Thread (sr, "Thread che saluta");
```

con una successiva chiamata del metodo `getName()` è possibile conoscere il nome del thread.

In ogni caso se non è stato assegnato al momento della creazione, il runtime Java provvede ad assegnare a ciascun thread un nome simbolico che però non è molto esplicativo all'occhio dell'utente.

L'uso di nomi significativi è particolarmente utile in fase di debugging, rendendo molto più facile individuare e selezionare il thread che si vuol porre sotto osservazione.

Maggior controllo sui thread

Oltre alla gestione ordinaria dei thread, Java fornisce una serie di strumenti che permettono di gestire l'esecuzione di un thread fin nei minimi dettagli. Se da un lato questo permette una maggiore capacità di controllo del thread stesso, dall'altro comporta un miglior controllo sulle risorse che sono utilizzate durante l'esecuzione.

Di conseguenza migliora il livello di portabilità della applicazione, dato che si può sopperire a certe carenze del sistema operativo.

Una fine tranquilla: uscire da `run()`

Negli esempi precedenti sono stati presi in considerazione casi con thread che eseguono un certo compito per un lasso limitato di tempo (stampare un certo messaggio 100 volte). Finito il compito, il thread termina e scompare dalla circolazione.

Spesso accade però che un thread possa vivere per tutta la durata dell'applicazione e svolgere il suo compito indefinitamente, senza mai terminare; oppure continui finché il suo funzionamento non venga fatto cessare volutamente.

Un esempio tipico potrebbe essere quello che segue: un "thread-orologio" mostra in questo caso l'ora corrente aggiornandola periodicamente:

```
public class Clock extends Applet implements Runnable {
   private Thread clockThread = null;

   public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");
         clockThread.start();
      }
   }

   public void run() {
      while (clockThread != null) {
         repaint();
      try {
         // rimane in attesa per un secondo
         Thread.sleep(1000);
      }
      catch (InterruptedException e) {}
      }
   }

   public void paint(Graphics g) {
      // prende data e ora corrente
      Calendar systemTime = Calendar.getInstance();
      // formatta e visualizza l'ora
      DateFormat formatter = new SimpleDateFormat("HH:mm:ss");
      g.drawString(formatter.format(systemTime.getTime()), 5, 10);
   }

   public void stop() {
      clockThread = null;
   }
}
```

Al momento della creazione e inizializzazione della Applet (alla visualizzazione della pagina HTML nel browser) viene creato e fatto partire un thread.

Quando la pagina che contiene l'Applet viene lasciata, viene eseguito il metodo `stop()`, che mette la variabile `clockThread` a `null`. Ciò ha un doppio effetto: determina la terminazione del thread (quando la variabile ha il valore `null` il ciclo `while` del metodo `run` ha termine), e rende disponibile la memoria dell'oggetto `Thread` per la garbage collection.

Bisogno di riposo: il metodo `sleep()`

Riprendendo in esame la classe `Clock`, si può notare che nel metodo `run()` viene chiamato il metodo `repaint()`, che a sua volta determina l'esecuzione del metodo `paint()`, il quale infine visualizza l'ora di sistema.

Se non si utilizzassero ulteriori accorgimenti si avrebbe un grande spreco di risorse e un funzionamento tutt'altro che ottimale: l'ora infatti sarebbe continuamente aggiornata, senza alcuna utilità dal momento che vengono visualizzati solo i secondi, causando per di più un rallentamento del sistema e uno sfarfallio dell'immagine (effetto flickering).

Per evitare tutto questo viene in aiuto il metodo `sleep()`, che permette di sospendere l'esecuzione di un thread (facendolo passare allo stato blocked) per un periodo di tempo prefissato, specificato in millisecondi. Nel caso in questione, una sospensione della durata di un secondo è esattamente l'intervallo sufficiente per l'aggiornamento dell'orologio.

Una chiamata del metodo `sleep()` provoca una messa in attesa del thread corrente e l'esecuzione del primo thread in attesa, con conseguente cambio di contesto, non previsto dalla tabella dello scheduler. Questa operazione da una parte comporta un certo costo computazionale (che va tenuto presente), ma dall'altra libera una risorsa che talvolta, come nel nostro esempio, rimarrebbe inutilmente occupata.

Si tenga presente che la sospensione del thread per mezzo di una `sleep` può essere pericolosa nel caso in cui si implementi una qualche gestione sincronizzata delle variabili (vedi oltre), dato che non rilascia gli eventuali lock acquisti dal thread.

Il metodo `sleep()` è un metodo statico della classe `Thread`, e ha come effetto di sospendere l'esecuzione del thread corrente. Di conseguenza è possibile chiamarlo da qualunque classe, anche se non viene usato alcun oggetto di tipo `Thread`.

Gioco di squadra: il metodo `yield()`

Si è visto precedentemente, parlando dei processi, che esiste una forma di multitasking chiamato *cooperativo*, in cui ogni processo cede volontariamente il controllo del processore, dato che il sistema non gestisce lo scheduling dei processi. Si è anche detto che le specifiche della virtual machine non prevedono il time-slicing per cui, in presenza di thread di uguale priorità non è garantito che un thread che non rilasci le risorse di sua iniziativa non resti in esecuzione indefinitamente, impedendo di fatto agli altri thread di funzionare.

Per questi motivi, normalmente è buona norma non definire blocchi di istruzioni che possono richiedere molto tempo per essere eseguite ma, in alternativa, spezzare tali blocchi in entità più piccole. Lo scopo è quello di facilitare il compito dell'algoritmo di "schedulazione" in modo da evitare che un solo thread monopolizzi il processore per periodi troppo lunghi.

Anche nel caso in cui il sistema si faccia carico di partizionare il tempo di esecuzione, spesso lo scheduler non è in grado di stabilire in maniera automatica dove e quando risulti più opportuno interrompere un thread.

Il metodo `yield()` permette di gestire in maniera ottimale queste situazioni: esso consente infatti di cedere l'uso del processore a un altro thread in attesa con il grosso vantaggio che, nel caso in cui nessuno sia in attesa di essere servito, permette il proseguimento delle operazioni del thread invocante senza un inutile e costoso cambio di contesto.

L'invocazione di `yield()` non provoca un cambio di contesto (il thread rimane runnable), ma piuttosto viene spostato alla fine della coda dei thread della sua stessa priorità.

Ciò significa che questo metodo ha effetto solo nei confronti di altri thread di uguale priorità, dato che i thread a priorità inferiore non prendono il posto del thread corrente anche se questo usa il metodo `yield()`.

Utilizzando `yield` è il programmatore che stabilisce come e dove è opportuno cedere il processore, indipendentemente da quello che è poi il corso storico dei vari thread.

È bene eseguire una chiamata a tale funzione in quei casi in cui si ritiene che il thread possa impegnare troppo a lungo il processore, in modo da facilitare la cooperazione fra thread, permettendo una migliore gestione delle risorse condivise.

Il metodo `yield()` è un metodo statico della classe `Thread`, e ha effetto sul thread corrente. È possibile quindi chiamarlo da qualunque classe senza riferimento a un oggetto di tipo `Thread`.

La legge non è uguale per tutti: la priorità

Si è visto che la virtual machine adotta uno scheduling di tipo preemptive, basato sulla priorità: ogni volta quindi che un thread di priorità maggiore del thread in esecuzione diventa runnable, si ha un cambio di contesto; per questo in linea di massima il thread corrente è sempre un thread a priorità più alta.

Si è anche detto che la virtual machine non prevede necessariamente il time-slicing ma, se questo è presente, i thread a maggiore priorità dovrebbero occupare la CPU per un tempo maggiore rispetto a quelli a minore priorità.

L'esempio che segue mostra questi aspetti dei thread, illustrando l'uso dei metodi `setPriority()` e `getPriority()`; la classe `CounterThread` rappresenta il thread di base, utilizzato in seguito dalla `ThreadPriority`.

```
private class CounterThread extends Thread {
   boolean terminated = false;
   int count = 0;

   public void run() {
      while (!terminated) {
         count++;
         for (int i = 0; i < 1000; i++) {
```

```
            ;
        }
    }

    public void terminate() {
        terminated = true;
    }

    public int getCount() {
        return count;
    }
}
```

La classe che implementa `Runnable`, oltre ad utilizzare il thread precedente, imposta anche le priorità.

```
public class ThreadPriority implements Runnable {
    CounterThread thread1 = new CounterThread();
    CounterThread thread2 = new CounterThread();
    Thread thisThread = new Thread(this);
    int duration;

    public ThreadPriority(int priority1, int priority2,
                          int duration) {
        this.duration = duration;
        thisThread.setPriority(Thread.MAX_PRIORITY);
        thread1.setPriority(priority1);
        thread2.setPriority(priority2);
        thread1.start();
        thread2.start();
        thisThread.start();
    }


    public void run() {
        try {
            for (int i = 0; i < duration; i++){
                System.out.println("Thread1: priority: "
                                   + thread1.getPriority()
                                   + " count: " + thread1.count);
                System.out.println("Thread2: priority: "
                                   + thread2.getPriority()
                                   + " count: " + thread2.count);
                thisThread.sleep(1000);
            }
        }
        catch (InterruptedException e){}

        thread1.terminate();
```

```
      thread2.terminate();

   }

   public static void main(String[] args) {
      new ThreadPriority(Integer.parseInt(args[0]),
                         Integer.parseInt(args[1]),
                         Integer.parseInt(args[2]));
   }
}
```

La classe `ThreadPriority` crea due oggetti `CounterThread` e li manda in esecuzione; successivamente manda in esecuzione il suo thread collegato (si tratta di una classe che implementa `Runnable`), il quale stampa i valori delle priorità e del counter dei thread ogni secondo (scheduler permettendo), e alla fine termina i due thread.

I valori di priorità e la durata in secondi sono dati come argomenti del main sulla linea di comando. I valori di priorità devono essere numeri interi da 1 a 10.

Si può notare che si è assegnata una priorità massima a `thisThread`, che deve poter interrompere gli altri due thread per eseguire la stampa e le chiamate `terminate()`: per fare questo si è usata la costante `Thread.MAX_PRIORITY`, che ha un valore uguale a 10.

La classe `CounterThread` aggiorna un contatore dopo aver eseguito un ciclo vuoto di 1000 iterazioni (ovviamente consumando una quantità abnorme di tempo della CPU, ma ai fini dell'esempio sorvoliamo su quest'aspetto).

Mandando in esecuzione il programma si può notare che effettivamente dopo un certo tempo il programma termina e vengono stampate le informazioni, il che significa che i due `CounterThread` sono stati interrotti dall'altro thread a priorità massima.

Se il sistema operativo e la VM supportano il time-slicing, il numero raggiunto dal contatore è approssimativamente proporzionale alla priorità del thread. Bisogna tener presente che possono esserci variazioni anche notevoli dato che i thread possono essere gestiti secondo algoritmi abbastanza complessi e variabili da implementazione a implementazione.

Tuttavia si nota che comunque, aumentando la priorità, aumenta il valore del contatore, e viceversa.

L'uso della gestione diretta delle priorità risulta molto utile in particolare nei casi in cui si ha un thread che resta nello stato blocked per la maggior parte del tempo. Assegnando a questo thread una priorità elevata si evita che rimanga escluso dall'uso della CPU in sistemi che non utilizzano il time-slicing. Questo è particolarmente importante per operazioni temporizzate che devono avere una certa precisione.

In casi in cui un certo thread compie delle operazione che fanno un uso intenso della CPU, e di lunga durata, abbassando la priorità del thread si disturba il meno possibile l'esecuzione degli altri thread. Ciò, ovviamente, sempre a patto che sia possibile mettere in secondo piano tale task.

E l'ultimo chiuda la porta: il metodo `join()`

Il metodo `join` resta semplicemente in attesa finché il thread per il quale è stato chiamato non termina la sua esecuzione. Con questo metodo è quindi possibile eseguire una determinata operazione nel momento in cui un thread termina la sua esecuzione. Risulta pertanto utile in tutti i casi in cui un thread compie delle operazioni che utilizzano dei risultati dell'elaborazione di un altro thread.

Di seguito è riportato un breve esempio nel quale è mostrato come utilizzare tale metodo. In esso si è utilizzata una versione modificata della classe `CounterThread`, in cui è possibile specificare, come parametro del costruttore, il massimo valore raggiungibile dal counter. In tal modo possiamo limitare la durata di esecuzione del thread senza dover ricorrere al metodo `terminate()`.

```
private class CounterThread extends Thread {
   boolean terminated = false;
   int count = 0;
   int maxCount = Integer.MAX_VALUE;

   public CounterThread() {
   }

   public CounterThread(int maxCount) {
      this.maxCount = maxCount;
   }

   public void run() {
      while (!terminated) {
         count++;
         for (int i = 0; i < 1000; i++) {
            // fai qualcosa
      }
   }

   public void terminate() {
      terminated = true;
   }

   public int getCount() {
      return count;
   }
}
```

La classe `Chronometer` misura il tempo di esecuzione, in minuti, secondi e millisecondi, di un thread che viene dato come argomento al metodo `run()`. Il metodo `join()` consente di determinare l'istante in cui termina l'esecuzione del thread (ovviamente con una certa approssimazione), e quindi di misurare il tempo trascorso dall'inizio dell'esecuzione.

```
public class Chronometer{
   Calendar startTime;
   Calendar endTime;

   public void run(Thread thread) {
      // registra l'ora di sistema all'inizio dell'esecuzione
      startTime = Calendar.getInstance();
      // manda in esecuzione il thread
      thread.start();
      try {
         // attende la fine dell'esecuzione
         thread.join();
      }
      catch (InterruptedException e) {}
      // registra l'ora di sistema alla fine dell'esecuzione
      endTime = Calendar.getInstance();
   }

   // calcola il tempo trascorso e restituisce
   // una stringa descrittiva
   public String getElapsedTime() {
      int minutes = endTime.get(Calendar.MINUTE)
                  - startTime.get(Calendar.MINUTE);
      int seconds = endTime.get(Calendar.SECOND)
                  - startTime.get(Calendar.SECOND);
      if (seconds < 0) {
         minutes--;
         seconds += 60;
      }
      int milliseconds = endTime.get(Calendar.MILLISECOND)
                       - startTime.get(Calendar.MILLISECOND);
      if (milliseconds < 0) {
         seconds--;
         milliseconds += 1000;
      }

return Integer.toString(minutes) + " minuti, " + seconds + " secondi, "
                                  + milliseconds + " millisecondi";
   }

   public static void main(String[] args) {
      Chronometer chron = new Chronometer();
      // manda in esecuzione il thread per mezzo di Chronometer
      // dando come parametro al costruttore il numero massimo
      // raggiungibile dal counter, ricevuto a sua volta
      // come parametro di main, dalla linea di comando
      chron.run(new CounterThread(Integer.parseInt(args[0])));

      // stampa il tempo trascorso
```

```
        System.out.println(chron.getElapsedTime());
    }
}
```

I metodi `sleep()` e `join()` sono metodi che hanno in comune la caratteristica di mettere un thread in stato di attesa (blocked). Ma mentre con `sleep()` l'attesa ha una durata prefissata, con `join()` l'attesa potrebbe protrarsi indefinitamente, o non avere addirittura termine. Alcune volte il protrarsi dell'attesa oltre un certo limite potrebbe indicare un malfunzionamento o comunque una condizione da gestire in maniera diversa che standoo semplicemente ad aspettare.

In questi casi si può usare il metodo `join(int` *`millisecondi`*`)` che permette di assegnare un limite massimo di attesa, dopo in quale il metodo ritornerà comunque, consentendo al metodo chiamante di riprendere l'esecuzione.

Sia `sleep()` che `join()` mettono in attesa il thread corrente, ma il metodo `join()` non è un metodo statico: viene chiamato per un oggetto specifico, che è quello di cui si attende la terminazione.

Interruzione di un thread

Un'altra caratteristica che accomuna thread e processi, è quella di essere soggetti a interruzioni. Come si è visto, l'interruzione è legata a un evento particolare, in qualche modo eccezionale, che determina cambiamenti tali nel contesto dell'esecuzione da richiedere (o poter richiedere) una gestione particolare dell'evento, ossia l'esecuzione di un codice specifico che fa fronte all'evento occorso.

In un thread l'interruzione ha luogo quando da un altro thread viene chiamato il metodo `interrupt()` per quel thread; formalmente è vero che un thread potrebbe interrompere se stesso, ma la cosa avrebbe poco senso.

L'aspetto più rilevante di questo metodo è che è in grado di interrompere uno stato di attesa causato da una chiamata a `sleep()` o `join()` (il discorso vale anche per il metodo `wait()` di cui si parlerà in seguito).

Se si ripensa per un momento agli esempi precedenti in cui vengono usati questi metodi, si nota che le chiamate sono all'interno di un blocco `try` e che nel `catch` viene intercettata un'eccezione del tipo `InterruptedException` anche se in questi casi l'eccezione non viene gestita.

Questo è appunto l'effetto di una chiamata al metodo `interrupt()`: se il thread interrotto è in stato di attesa, viene generata un'eccezione del tipo `InterruptedException` e lo stato di attesa viene interrotto.

L'oggetto `Thread` ha così l'opportunità di gestire l'interruzione, eseguendo del codice all'interno del blocco `catch`. Se il blocco `catch` è vuoto, l'effetto dell'interruzione sarà semplicemente quello di far riprendere l'esecuzione (non appena il thread, passato nuovamente allo stato runnable, sarà mandato in esecuzione dallo scheduler) dall'istruzione successiva alla chiamata `sleep()` o `join()`.

Cosa accade se invece viene interrotto un thread che non è in attesa? In questo caso viene modificata una variabile di stato del thread facendo sì che il metodo `isInterrupted()` restituisca `true`. Questo permette al thread di gestire ugualmente l'interruzione controllando (tipicamente alla fine o comunque all'interno di un ciclo) il valore restituito da questo metodo:

```
public void run() {
   while (true) {
      doMyJob();
      if (isInterrupted())
         handleInterrupt();
   }
}
```

Purtroppo il flag di interruzione non viene impostato se l'interruzione ha luogo durante uno stato di attesa, per cui un codice del genere non funzionerebbe correttamente:

```
public void run() {
   while (true) {
      doMyJob();

      try {
         sleep(100);
      } catch (InterruptedException e) {}

      if (isInterrupted())
         handleInterrupt();
   }
}
```

Infatti, se l'interruzione ha luogo durante l'esecuzione di `sleep()`, viene generata una eccezione, ma il metodo `isInterrupted()` restituisce `false`.

Se si vuole gestire l'interruzione indipendentemente dal momento in cui si verifica, bisogna duplicare la chiamata a `handleInterrupt()`:

```
public void run() {
   while (true) {
      doMyJob();

      try {
```

```
            sleep(100);
        } catch (InterruptedException e) {
            handleInterrupt()
        }

        if (isInterrupted())
            handleInterrupt();
    }
}
```

Il metodo `interrupt()` generalmente non interrompe un blocco dovuto ad attesa di I/O. In questi casi si deve agire direttamente sugli stream per interrompere lo stato di attesa. Il metodo `interrupt` è stato introdotto con Java 1.1 e non funziona con Java 1.0. Inoltre spesso non è supportato dalla VM dei browser, anche di quelli che dovrebbero supportare Java 1.1. Quindi per ora è opportuno evitarne l'uso nelle applet, a meno che non si faccia uso del Java plug-in.

Metodi deprecati

Il metodo `stop()`, che termina l'esecuzione di un thread, oltre a essere stato deprecato in Java 2, è sconsigliato: infatti il suo uso rischia di produrre malfunzionamenti causando una interruzione "al buio" (cioè senza che il thread interrotto abbia il controllo delle modalità di terminazione. Per motivi analoghi sono deprecati i metodi `suspend()`, che mette il thread nello stato blocked, e `resume()` che lo sblocca, riportandolo allo stato runnable.

La sincronizzazione dei thread

Parlando dei processi, nell'introduzione, si è detto che questi hanno spazi di memoria separati e che possono condividere e scambiare dati tra loro solo con mezzi particolari appositamente concepiti per questo scopo. Si è inoltre detto che i thread che appartengono al medesimo processo condividono automaticamente lo spazio di memoria.

Come si è visto, una classe `Thread` o una basata sul `Runnable` funzionano come normali classi, e come tali hanno accesso a tutti gli oggetti che rientrano nel loro scope.

La differenza fondamentale è che, mentre le normali classi funzionano una alla volta, ossia eseguono il loro codice in momenti differenti, i thread vengono eseguiti in parallelo; questo significa che esiste la possibilità che thread diversi accedano contemporaneamente agli stessi dati. Anche se per "contemporaneamente" si intende sempre qualcosa basato su un parallelismo simulato, vi sono casi in cui questa "simultaneità" di accesso, per quanto relativa, può causare effettivamente dei problemi.

Sorge così l'esigenza di implementare una qualche tecnica di sincronizzazione dei vari thread. Prima di spiegare nei dettagli gli aspetti legati alla sincronizzazione, si ponga attenzione a i modi in cui diversi thread condividono oggetti e dati.

Condivisione di dati fra thread

Il caso più comune è quello di oggetti creati esternamente che vengono passati come parametri a un oggetto `Thread` o `Runnable`. Ad esempio:

```
public class PointXY {

   int x;
   int y;

   public PointXY(int x, int y) {
      this.x = x;
      this.y = y;
   }
}

public class PointThread extends Thread {

   PointXY point;

   public Run1(PointXY p) {
      point = p;
   }

   public void run() {
      // esegue operazioni con la variabile point
   }
}

public class PointRunnable implements Runnable {

   PointXY point;

   public Run2(PointXY p) {
      point = p;
   }

   public void run() {
      // esegue operazioni con la variabile point
   }
}

...
```

```
// dall'esterno si istanziano e si lanciano
// i thread relativi Run1 e Run2
PointXY point = new PointXY(10, 10);
PointThread thread1 = new PointThread(point);
PointRunnable runnable = new PointRunnable(point);
Thread thread2 = new Thread(runnable);

// start ed utilizzazione...
```

In questo modo si è permessa una condivisione di una variabile tra due thread, di cui uno funziona come sottoclasse di `Thread`, l'altro è collegato a una `Runnable`. Tutti e due gli oggetti hanno accesso alla stessa istanza dell'oggetto `p` di tipo `PointXY`.

Si sarebbero potute anche creare due istanze della stessa classe:

```
// condivisione di uno stesso oggetto point
// tra due istanze di uno stessa classe Thread
PointThread thread1 = new PointThread(point);
PointThread thread2 = new PointThread(point);
```

oppure due `Thread` collegati allo stesso `Runnable`:

```
// condivisione di uno stesso oggetto Runnable
// da parte di due Thread e di conseguenza
// condivisione dello stesso oggetto point
PointRunnable runnable = new PointRunnable(point);
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
```

In questi due casi i thread non solo condividono l'oggetto `point`, ma eseguono anche lo stesso codice in maniera indipendente ed eventualmente con differenti modalità. Nell'ultimo caso si è in presenza di una sola istanza di un oggetto `Runnable`, a cui si passa l'oggetto `point`, che viene "agganciato" a due thread differenti, mentre nel primo caso si creano due istanze di una sottoclasse di `Thread`, a cui si passa lo stesso oggetto `point`.

Competizione fra thread

Dopo aver accennato ad una delle configurazioni tipiche di accesso concorrente ad aree di memoria, si può passare a considerare quali siano i potenziali problemi derivanti dalla condivisione dei dati e dall'accesso parallelo a questi dati.

Si consideri ad esempio l'interfaccia `Value` che funge da contenitore (wrapper) di un valore intero

```
public interface Value {
   public abstract int get();
```

```
   public abstract void set(int i);
   public abstract void increment();
}
```

La classe `IntValue` implementa l'interfaccia di cui sopra fornendo una gestione del valore contenuto come intero

```
public class IntValue implements Value {
   int value = 0;

   public int get() {
      return value;
   }

   public void set(int i) {
      value = i;
   }

   public void increment() {
      value++;
   }
}
```

La classe `StringValue` invece fornisce una gestione del valore come stringa

```
public class StringValue implements Value {
   String value = "0";

   public int get() {
      return Integer.parseInt(value);
   }

   public void set(int i) {
      value = Integer.toString(i);
   }

   public void increment() {
      int i = get();
      i++;
      set(i);
   }
}
```

Infine il thread permette di incrementare il valore contenuto in un generico wrapper, che viene passato al costruttore come interfaccia generica

```
public class ValueIncrementer extends Thread {
   Value value;
```

```
  int increment;

  public ValueIncrementer(Value value, int increment) {
    this.value = value;
    this.increment = increment;
  }

  public void run() {
    for (int i = 0; i < increment; i++)
      value.increment();
  }

  public static void main(String[] args) {
    // crea un IntValue
    IntValue intValue = new IntValue();
    // crea due IntIncrementer a cui passa lo stesso IntValue
    // e lo stesso valore di incremento pari a 100000
    ValueIncrementer intIncrementer1
     = new ValueIncrementer(intValue, 100000);
    ValueIncrementer intIncrementer2
     = new ValueIncrementer(intValue, 100000);
    // ripete i passi precedenti
    // questa volta con un oggetto StringValue
    StringValue stringValue = new StringValue();
    ValueIncrementer stringIncrementer1
     = new ValueIncrementer(stringValue, 100000);
    ValueIncrementer stringIncrementer2
     = new ValueIncrementer(stringValue, 100000);

    try {
      // fa partire insieme i due thread che
      // incrementano lo IntValue
      intIncrementer1.start();
      intIncrementer2.start();
      // attende che i due thread terminino l'esecuzione
      intIncrementer1.join();
      intIncrementer2.join();
      // stampa il valore
      System.out.println("int value: " + intValue.get());

      // ripete i passi precedenti
      // questa volta con lo StringValue
      stringIncrementer1.start();
      stringIncrementer2.start();
      stringIncrementer1.join();
      stringIncrementer2.join();
      System.out.println("string value: " + stringValue.get());
    } catch (InterruptedException e) {}
  }
}
```

Tralasciando le considerazioni legate al modo in cui si effettua l'incremento, mandando in esecuzione l'esempio (impiegando una VM che utilizza il time-slicing), si può notare che mentre il valore per l'oggetto di tipo `IntValue` è quello che ci si aspetta, dovuto all'incremento di 100000 effettuato da due thread distinti, il valore dell'oggetto `StringValue` è inferiore, e varia da esecuzione a esecuzione.

Per capire cosa sia successo si esamini il codice del metodo `increment()` delle due classi `IntValue` e `StringValue`. Nella classe `IntValue` si ha

```
public void increment() {
   value++;
}
```

ovvero il metodo compie una semplice operazione di incremento di una variabile di tipo `int`. Invece, nella classe `StringValue` si trova

```
public void increment() {
   int i = get();
   i++;
   set(i);
}
```

Qui siamo in presenza di un algoritmo che, per quanto semplice, è formato da diverse istruzioni; a loro volta i metodi `get()` e `set()` chiamano metodi della classe `Integer` per convertire la stringa in `int` e viceversa, metodi che compiono operazioni di una certa complessità, ossia eseguono diverse istruzioni, ma possiamo anche prescindere da quest'ultima osservazione. Quello che conta è che si tratta comunque di un'operazione complessa, divisa in più passi successivi. È questa complessità dell'operazione ciò che causa il problema. Infatti, se i due thread funzionano in parallelo in un sistema gestito con il time-slicing, è possibile che il passaggio da un thread all'altro avvenga durante l'esecuzione del metodo `increment()`.

In questo caso, dato che i thread non fanno altro che eseguire `increment()` in un ciclo, e hanno la stessa priorità, le probabilità sono piuttosto alte. Di conseguenza, è possibile che il processore segua una sequenza di esecuzione come questa (per semplicità le chiamate `get()` e `set()` saranno ipotizzate istruzioni semplici):

1. Il primo thread esegue l'istruzione `int i = get();`
   Supponendo che il valore sia `100`, questo valore viene assegnato alla variabile locale `i`

2. Il secondo thread esegue l'istruzione `int i = get();`
   Il valore è sempre `100`, e viene assegnato all'altra variabile locale `i` (che è diversa per ciascun thread)

3. Il primo thread esegue l'istruzione `i++;`

4. Il valore della variabile locale diventa `101`
   Il primo thread esegue l'istruzione `set(i);`
   La variabile di classe (condivisa fra i thread) diventa `101`

5. Il secondo thread esegue l'istruzione `i++;`
   Il valore della variabile locale passa da `100` a `101`

6. Il secondo thread esegue l'istruzione `set(i);`
   Alla variabile di classe viene assegnato il valore `101`, che è lo stesso che già aveva, in seguito all'azione del primo thread.

Quindi, come risultato complessivo si ottiene un incremento di `1` e non di `2`, come se uno dei due thread non avesse fatto nulla. Questa situazione di interferenza tra thread è quella che viene generalmente chiamata *race condition*.

Si noti come nel caso degli oggetti `IntValue` non si sia verificata nessuna alterazione: infatti il metodo `increment()` di questa classe compie una sola operazione non complessa, l'incremento della variabile interna, ossia un'istruzione che non può essere interrotta nel mezzo da un cambio di contesto. Le operazioni di questo tipo sono chiamate *atomiche*.

Alcune operazioni in apparenza atomiche, possono in realtà non esserlo: infatti se una variabile di tipo int è un rappresentata da un'area indivisibile di 32 bit, e quindi un'operazione di scrittura viene eseguita in una sola operazione non interrompibile, altrettanto non vale per un long, che occupa 64 bit di memoria, e in certi casi viene scritto o letto in due blocchi di 32 bit con due operazioni distinte. Se fra una operazione e la successiva si interrompe il thread, ci può essere un'alterazione non voluta del valore della variabile. Si tenga presente che la frammentazione di una operazione di scrittura in sottoperazioni avviene a basso livello in maniera non visibile dal programma Java. Una race condition (condizione di competizione) è la situazione che si verifica quando due o più thread eseguono contemporaneamente operazioni di cui almeno una non atomica sugli stessi dati; l'ordine con il quale i vari passi di cui le operazioni sono composte vengono eseguiti dai diversi thread può portare ad alterazioni del risultato dell'intera operazione per uno o più thread.

Lock e sincronizzazione

Per risolvere una simile situazione è necessario che l'operazione complessa sia effettuata per intero da un thread alla volta, e che non venga interrotta da istruzioni relative alla

stessa operazione eseguite da un altro thread. Per ottenere questo risultato generalmente si ricorre all'utilizzo dei cosiddetti lock.

Il lock può essere paragonato alla chiave di una toilette: alla toilette accede una sola persona alla volta e una volta entrata chiude la porta a chiave, dato che, anche in questo caso, sia pure per motivi differenti rispetto al caso dei thread, la condivisione della risorsa potrebbe portare a risultati indesiderati. Le altre persone che vogliono entrare, trovano la porta chiusa e devono pertanto attendere l'uscita dell'utente corrente della toilette.

Il lock può essere pensato come una semplice variabile booleana, visibile da tutti i thread. Ogni volta che un thread esegue un codice protetto da un lock, la variabile viene impostata a `true`, per indicare che il codice è già in esecuzione, e si dirà che il thread ha acquisito il lock su quel codice.

Il meccanismo di "schedulazione" si fa carico di garantire che, fin quando un thread è in possesso di un lock su un certo codice, nessun altro thread vi acceda. Eventuali thread che chiedono l'accesso al codice vengono così messi in attesa finché il thread corrente non ha rilasciato il lock.

In Java, però, tutto questo avviene "dietro le quinte", dal momento che il programmatore non usa direttamente dei lock, ma ricorre invece alla keyword `synchronized`. Il meccanismo è estremamente semplice.

Si riconsideri l'esempio di cui sopra facendo una piccola ma importante modifica al metodo `increment()`

```
public synchronized void increment() {
    int i = get();
    i++;
    set(i);
}
```

Se si prova adesso a eseguire l'applicazione, si potrà vedere che il risultato è corretto, per entrambe le classi `IntValue` e `StringValue`.

Quando un thread esegue un metodo `synchronized`, acquisisce il lock prima di eseguire le istruzioni, e lo rilascia al termine dell'esecuzione.

Ma acquisisce il lock su che cosa? La risposta è: acquisisce il lock sull'oggetto stesso. Più precisamente quando un oggetto è sottoposto a lock, nessuno dei suoi metodi sincronizzati è eseguibile se non dal thread che detiene il lock (cosa molto importante al fine di evitare deadlock).

Per capire meglio cosa questo significhi in pratica, si consideri una nuova classe `ValueIncrementer2` identica alla classe `ValueIncrementer`, ma con una piccola modifica nel metodo `run()`:

```
public void run() {
    for (int i = 0; i < increment; i++)
```

```
        value.set(value.get() + 1);
  }
```

In questo caso l'incremento non è più ottenuto con una chiamata al metodo `increment()`, ma chiamando direttamente i metodi `get()` e `set()`.

Se si prova ad eseguire contemporaneamente un oggetto `ValueIncrementer` e un oggetto `ValueIncrementer2`, nonostante la sincronizzazione del metodo `increment()` della classe `StringValue`, si otterrà una race condition con forti probabilità di funzionamento anomalo.

Il motivo di questa incomprensibile stranezza risiede nel fatto che `increment()` è l'unico metodo sincronizzato: ciò implica non solo che sia l'unico ad acquisire il lock, ma anche che sia l'unico a rispettarlo. In sostanza il lock non ha alcun effetto sui metodi non sincronizzati, in particolare sui metodi `get()` e `set()`, che quindi possono essere eseguiti in parallelo a `increment()` e causare i problemi che abbiamo visto.

Per evitare questi problemi, si devono definire come `synchronized` anche i metodi `get()` e `set()`. In tal modo, poiché il lock è sull'oggetto, sarà impossibile mandare contemporaneamente in esecuzione due metodi sincronizzati dello stesso oggetto; nel nostro caso i metodi `increment()`, `get()` e `set()` non potranno essere eseguiti in parallelo sullo stesso oggetto `StringValue`, ma dovranno attendere ognuno la fine dell'esecuzione dell'altro su un altro thread. Infine si tenga presente che una variabile non può essere direttamente sottoposta a lock, dato che si possono sincronizzare solo i metodi.

Quindi per permettere realmente la sincronizzazione sull'accesso concorrente a una variabile, oltre a definire sincronizzati tutti i metodi di gestione di tale variabile, si dovrà impedire l'accesso diretto per mezzo di una istruzione del tipo

```
oggetto.variabile = valore
```

Quindi tutte le variabili passibili di accesso condiviso devono essere protette e ad esse si deve accedere esclusivamente con metodi sincronizzati pubblici.

Un oggetto si dice thread-safe quando è protetto da malfunzionamenti causati da race condition e quindi è correttamente eseguibile anche contemporaneamente da thread differenti. Lo stesso termine può essere riferito anche a singoli metodi o a intere librerie.

Visibilità del lock

L'uso di `synchronized` fino ad ora è stato applicato a un intero metodo. Esiste anche la possibilità di circoscrivere l'ambito della sincronizzazione a un blocco di codice, otte-

nendo così un blocco sincronizzato. Ecco un'altra versione del metodo `increment()` di `StringValue`, che esemplifica questa modalità:

```
public void increment() {
   synchronized (this) {
      int i = get();
      i++;
      set(i);
   }
}
```

In questo caso le due versioni del metodo sono esattamente equivalenti, dato che il lock viene acquisito all'inizio del metodo e rilasciato alla fine.

Può capitare però che solo una porzione di codice all'interno di un metodo necessiti di sincronizzazione. In questi casi può essere opportuno usare un blocco sincronizzato piuttosto che un metodo sincronizzato, restringendo lo scope del lock.

Per visibilità di un lock (scope del lock) si intende il suo ambito di durata, corrispondente alla sequenza di istruzioni che viene eseguita tra il momento in cui il lock viene acquisito e quello in cui viene rilasciato.

Utilizzando il blocco sincronizzato si deve anche specificare l'oggetto di cui vogliamo acquisire il lock. Nell'esempio precedente l'oggetto è lo stesso di cui si sta eseguendo il metodo, e quindi è indicato con `this`, ma potrebbe essere anche un altro.

Questo significa che un lock su un oggetto può aver effetto anche su codice di altri oggetti, anche di classi differenti. Quindi, correggendo un'affermazione precedentemente fatta, dal contenuto ancora impreciso, si può dire che quando un oggetto è sottoposto a lock, nessuna area sincronizzata — intendendo sia blocchi che metodi — che richieda il lock per quel determinato oggetto è eseguibile se non dal thread che detiene il lock.

Tra brevissimo sarà preso in esame un esempio di uso di un blocco sincronizzato per un oggetto diverso da `this`. Ma quali sono i criteri in base ai quali scegliere lo scope appropriato? Bisogna naturalmente valutare caso per caso tenendo presente i diversi aspetti a favore e contro.

Da una parte, uno scope più esteso del necessario può causare inutili ritardi nell'esecuzione di altri thread, e in casi particolari può anche portare a una situazione di stallo, detta deadlock, di cui si dirà tra poco.

Dall'altra, acquisire e rilasciare un lock è un'operazione che consuma delle risorse e quindi, se si verifica troppo di frequente, rischia di influire negativamente sull'efficienza del programma. Inoltre, come vedremo, anche l'acquisizione di troppi lock può portare al verificarsi di deadlock.

Lo scope di un'area sincronizzata in Java non può estendersi al di là di un singolo metodo. Nel caso servano lock di scope più estesi (che vengano acquisiti in un metodo e rilasciati in un'altro, eventualmente di un'altro oggetto) occorre ricorrere a lock implementati *ad hoc* (ad esempio una classe `Lock`) la cui trattazione esula dagli scopi di questo capitolo.

Deadlock

Si supponga di scrivere una classe `FileUtility` che fornisca una serie di funzioni di utilità per il file system. Una delle funzioni è quella di eliminare da una directory tutti i files la cui data è precedente a una certa data fissata dall'utente, oppure esistenti da più di un certo numero di giorni. Un'altra funzione è di comprimere i files di una certa directory.

Si supponga di aver creato due classi:

la classe `File`, che tra l'altro contiene un metodo `isOlder(Date d)` che controlla se il file è antecedente a una certa data, e un metodo `compress()` che comprime il file;

la classe `Directory`, che contiene tra gli altri un metodo `removeFile(File f)`, e dei metodi `firstFile()` e `nextFile()` utilizzabili per iterare sui files della directory, che sono mantenuti come una collezione di oggetti `File` all'interno dell'oggetto `Directory`.

La classe `FileUtility`, da parte sua, contiene `removeOldFiles(Directory dir, Date date)`, un metodo che elimina i files "vecchi", e `compressFiles(Directory dir)`, un metodo che comprime tutti i file di una directory.

Questa potrebbe essere una implementazione del metodo `removeOldFiles`:

```
public void removeOldFiles (Directory dir, Date date) {
   for (File file = dir.firstFile(); file != null; dir.nextFile()) {
synchronized (file) {
         if (file.isOlder(date)) {
            synchronized (dir) {
               dir.removeFile(file);
            }
         }
      }
   }
}
```

Questo è un tipico esempio di uso del blocco sincronizzato su un oggetto diverso da `this`: quello che serve è un lock sul file, per evitare che altri thread possano agire contemporaneamente sullo stesso file. Se il file risulta "vecchio" si utilizza il metodo `removeFile()` dell'oggetto `Directory`, ed anche in questo caso si deve ottenere il lock

su tale oggetto, per evitare interventi contemporanei sulla stessa directory, che potrebbero interferire con l'operazione di cancellazione.

Questa potrebbe essere una possibile implementazione del metodo `compressFiles()`:

```
public void compressFiles (Directory dir) {
   synchronized (dir) {

for (File file = dir.firstFile(); file != null; dir.nextFile()) {
         synchronized (file) {
            file.compress();
         }
      }
   }
}
```

Anche in questa circostanza il thread deve acquisire i lock sull'oggetto `Directory` e sull'oggetto `File` per evitare interferenze potenzialmente dannose.

Si ipotizzi adesso che i due thread siano in esecuzione contemporaneamente e che si verifichi una sequenza di esecuzione come questa:

1. Il primo thread chiama il metodo `compressFiles()` per un certo oggetto `Directory`, acquisendone il lock;

2. Il secondo thread chiama il metodo `removeOldFiles()` per lo stesso oggetto `Directory`, verificando che il primo file è vecchio, e acquisisce il lock per il primo oggetto `File`;

3. Il secondo thread, per procedere alla rimozione del file, tenta di acquisire il lock sull'oggetto `Directory`, lo trova già occupato e si mette in attesa;

4. Il primo thread tenta di acquisire il lock per il primo oggetto `File`, lo trova occupato e si mette in attesa.

A questo punto i thread si trovano in una situazione di stallo, in cui ognuno aspetta l'altro, ma l'attesa non avrà mai termine. Si è verificato un deadlock.

Un deadlock è una situazione in cui due o più thread (o processi) si trovano in attesa l'uno dell'altro, in modo tale che gli eventi attesi non potranno mai verificarsi.

Per quanto riguarda la prevenzione dell'insorgenza di deadlock, non ci sono mezzi particolari messi a disposizione dal linguaggio, né regole generali e precise da seguire.

Si tratta di esaminare con attenzione le possibili interazioni fra thread e tenerne conto nell'implementazione delle classi. Ci sono naturalmente dei casi tipici, il cui esame va però al di là degli obiettivi di questo capitolo.

Class lock e sincronizzazione di metodi statici

La keyword `synchronized` può essere usata anche per metodi statici, ad esempio per sincronizzare l'accesso a variabili statiche della classe. In questo caso quello che viene acquisito è il lock della classe, anziché di un determinato oggetto di quella classe.

In realtà il lock si riferisce sempre a un oggetto, e precisamente all'oggetto `Class` che rappresenta quella classe. Quindi è possibile acquisire un lock della classe anche da un blocco sincronizzato, specificando l'oggetto `Class`:

```
public void someMethod() {
   synchronized (someObject.class) {
      doSomething();
   }
}
```

oppure:

```
public void someMethod() {
   synchronized (Class.forName("SomeClass")) {
      doSomething();
   }
}
```

Comunicazione fra thread

Dato che la programmazione per thread permette l'esecuzione contemporanea di più flussi di esecuzione autonomi fra loro, sorge abbastanza spontanea l'esigenza di mettere in comunicazione fra loro tali flussi in modo da realizzare qualche tipo di lavoro collaborativo. Il modo più semplice per ottenere la comunicazione fra thread è la condivisione diretta di dati, attraverso codice sincronizzato, come visto in precedenza. Ma ci sono situazioni in cui questo sistema non è sufficiente.

Condivisione di dati

Si consideri questo semplice esempio basato su le due classi `Transmitter` e `Receiver`, utilizzabili su thread differenti per lo scambio di dati:

```
public class Transmitter extends Thread {
```

```
   Vector data;

   public Transmitter(Vector v) {
      data = v;
   }

   public void transmit(Object obj) {
      synchronized (data) {
         data.add(obj);
      }
   }

   public void run() {
      int sleepTime = 50;
      transmit("Ora trasmetto 10 numeri");
      try {
         if (!isInterrupted()) {
            sleep(1000);
            for (int i = 1; i <= 10; i++) {
               transmit(new Integer(i * 3));
               if (isInterrupted())
                  break;
               sleep(sleepTime * i);
            }
         }
      } catch (InterruptedException e) {}
      transmit("Fine della trasmissione");
   }
}
```

La classe `Transmitter` implementa un semplice meccanismo di condivisione dei dati attraverso un oggetto `Vector`, che viene passato come argomento del costruttore. Il metodo `transmit()` non fa altro che aggiungere un elemento al `Vector`, dopo aver acquisito un lock sul `Vector` stesso.

Questa operazione ha un reale effetto perché la classe `Vector` è una classe thread-safe, ossia è stata implementata usando dove necessario dei blocchi o dei metodi sincronizzati.

Il metodo `run()` trasmette un messaggio iniziale, attende un secondo, poi trasmette una sequenza di 10 numeri a intervalli di tempo crescenti, infine trasmette un messaggio finale.

In questo metodo viene anche esemplificata una gestione delle interruzioni: i messaggi iniziale e finale vengono comunque trasmessi; in caso di interruzione durante la trasmissione viene conclusa la trasmissione in corso, poi si esce dal ciclo; se l'interruzione arriva durante una chiamata a `sleep()`, questa causa un salto al blocco `catch`, vuoto, con un risultato equivalente.

```
public class Receiver extends Thread {
   Vector data;

   public Receiver(Vector v) {
      data = v;
   }

   public Object receive() {
      Object obj;
      synchronized (data) {
         if (data.size() == 0)
            obj = null;
         else {
            obj = data.elementAt(0);
            data.removeElementAt(0);
         }
      }
      return obj;
   }

   public void run() {
      Object obj;
      while (!isInterrupted()) {
         while ((obj = receive()) == null) {
            try {
               sleep(1000);
            } catch (InterruptedException e) {
               return;
            }
         }
         System.out.println(obj.toString());
      }
   }
}
```

La classe `Receiver` riceve anch'essa un `Vector` come argomento del costruttore, e in tal modo ha la possibilità di condividere i dati con un `Transmitter`.

Il metodo `receive()` restituisce `null` se non trova dati; altrimenti restituisce il dato dopo averlo rimosso dal `Vector`.

Il metodo `run()` esegue un ciclo che può essere terminato solo da una chiamata a `interrupt()`.

Anche in questo caso viene gestito sia la possibilità di interruzione in stato di attesa con una `InterruptedException`, sia quella di interruzione durante l'esecuzione.

Ad ogni ciclo si prova a ricevere un dato: se la ricezione ha luogo, stampa il dato sotto forma di stringa, altrimenti attende un secondo e riprende il ciclo.

```
public class ThreadCommunication{
   public static void main(String[] args) {
      Vector vector = new Vector();
      Transmitter transmitter = new Transmitter(vector);
      Receiver receiver = new Receiver(vector);
      transmitter.start();
      receiver.start();
      try {
         transmitter.join();
         Thread.sleep(2000);
      } catch (InterruptedException e) {}
      receiver.interrupt();
   }
}
```

La classe `ThreadCommunication` contiene soltanto un `main()` che mostra il funzionamento delle classi `Transmitter` e `Receiver`: in tale metodo viene creato un oggetto di tipo `Vector` e lo passa ai due oggetti `Transmitter` e `Receiver`, che poi provvede a far partire.

Quando il `Transmitter` ha terminato la sua esecuzione, attende 2 secondi per dare al `Receiver` il tempo di ricevere gli ultimi dati; successivamente termina il `Receiver` con una chiamata al metodo `interrupt()`.

In questo caso il metodo `interrupt()` è usato per terminare il thread: questo è un sistema che può essere usato nei casi in cui non ci sia la necessità gestire le interruzioni diversamente, senza terminare il thread, ma eseguendo un determinato codice. La differenza, rispetto all'uso di un flag di stop, è che il thread termina immediatamente anche se si trova in stato di attesa.

Utilizzo dei metodi `wait()` e `notify()`

Per migliorare la sincronizzazione fra i thread, si può ricorrere all'utilizzo dei metodi `wait()` e `notify()`.

Si tratta di metodi appartenenti alla classe `Object` e non alla classe `Thread`, per cui possono essere utilizzati per qualunque oggetto. Il funzionamento è semplice: se un thread esegue una chiamata al metodo `wait()` di un determinato oggetto, il thread rimarrà in stato di attesa fino a che un altro thread non chiamerà il metodo `notify()` di quello stesso oggetto.

Con poche variazioni al codice, si possono modificare le classi appena viste in modo che utilizzino `wait()` e `notify()`.

Per prima cosa si può si può modificare il metodo `transmit()` della classe `Transmitter` in modo che sia effettuata una chiamata al metodo `notify()` per segnalare l'avvenuta trasmissione:

```
public void transmit(Object obj) {
   synchronized (data) {
      data.add(obj);
      data.notify();
   }
}
```

Nella classe `Receiver` invece il metodo `run()` diventa

```
public void run() {
   Object obj;
   while (!isInterrupted()) {
      synchronized (data) {
         while ((obj = receive()) == null) {
            try {
               data.wait();
            } catch (InterruptedException e) {
               return;
            }
         }
      }
      System.out.println(obj.toString());
   }
}
```

Come si può notare al posto della chiamata a `sleep()` si effettua una chiamata a `wait()`: in tal modo l'attesa si interromperà subito dopo la trasmissione, segnalata dalla chiamata `notify()`. Per il resto il funzionamento resta uguale.

Si può notare però un'altra differenza: all'interno del ciclo è stato inserito un blocco sincronizzato, pur essendo il metodo `receive()` già sincronizzato.

Il motivo è che utilizzando direttamente la `wait()` è necessario prima ottenere il lock sull'oggetto, altrimenti si causa un'eccezione del tipo `IllegalMonitorStateException`, che darà il poco intuitivo messaggio `current thread not owner`, che sta a significare che il thread non detiene il lock sull'oggetto.

Più precisamente questo significa che, essendo la `wait()` un metodo della classe `Object` e non della `Thread`, è possibile invocare una `wait()` su tutti gli oggetti Java, non solo sui thread, anche se l'invocazione può avvenire solo se preventivamente si è acquisito il lock relativo. Discorso del tutto analogo per il metodo `notify()`.

Questa è la tipica sequenza di eventi per la creazione di una sincronizzazione fra due thread; supponendo che il thread `thread1` chiami `wait()` e il thread `thread2` chiami `notify()`:

1. il `thread1` chiama `wait()` dopo aver acquisito il lock sull'oggetto; `wait()` per prima cosa rilascia il lock, poi mette il thread in attesa;

2. il `thread2` a questo punto può acquisire il lock ed eseguire il blocco sincronizzato da cui chiama `notify()` dopodiché il lock sarà rilasciato…

3. …da `thread1`; `wait()`, ricevuta la notifica riacquisisce il lock ed esce; successivamente l'esecuzione del codice potrà continuare.

Tipicamente questo tipo di comunicazione si usa per aspettare/notificare il verificarsi di una certa condizione.

Nell'esempio appena visto la condizione è la presenza di dati ricevuti; in mancanza del meccanismo di sincronizzazione descritto sopra, e permettendo l'uso di `wait()` e `notify()` al di fuori di aree sincronizzate, si potrebbe verificare una sequenza di questo tipo:

1. il `Receiver` controlla se ci sono dati ricevuti; non ne trova;

2. il `Transmitter` trasmette un dato,

3. il `Transmitter` chiama `notify()`, ma la notifica non arriva a destinazione, dal momento che non c'è ancora nessun thread in attesa;

4. il `Receiver` si mette in attesa, ma ormai la notifica è andata persa.

È importante osservare che se un thread acquisisce il lock su un determinato oggetto, solo esso potrà eseguire l'operazione di rilascio e, finché non effettuerà tale operazione (uscendo dal blocco sincronizzato), il lock risulterà sempre occupato.

Questo fatto ha un'importante conseguenza: il thread messo in stato di attesa, che restituisce temporaneamente il lock, potrà essere riattivato solo se, dopo la chiamata a `notify()`, viene ad esso restituito il lock sull'oggetto che originariamente aveva acquisito. Non è detto quindi che un thread riprenda immediatamente la sua esecuzione dopo una chiamata a `notify()`, ma può trascorrere un periodo di tempo non precisato.

Ad esempio, se si scrive

```
synchronized (object){
   doSomething();
   object.notify();
   doSomethingElse();
}
```

fino a che non viene terminata l'esecuzione di `doSomethingElse()`, il thread che ha chiamato `wait()` non può riprendere l'esecuzione, anche se ne è stata richiesta la riattivazione con `notify()`.

Il metodo `wait()` esiste anche nella versione `wait(int millisecondi)`: in questo caso viene specificato un timeout scaduto il quale lo stato di attesa termina anche se non è stata ricevuta alcuna notifica.

Questo metodo può essere usato al posto di `sleep()` quando si vuole bloccare momentaneamente il thread rilasciando contemporaneamente il lock acquisito.

Il metodo `notifyAll()`

Se i thread in accesso concorrente sono più di uno, e tutti in attesa a causa di una `wait()`, allora una chiamata alla `notify()` avvertirà uno solo dei thread in attesa, senza la possibilità di sapere quale. In situazioni del genere il metodo `notifyAll()`, permette di eseguire la notifica nei confronti di tutti i thread in attesa.

Nel caso in cui si desideri che sia solo un particolare thread tra quelli in attesa a riprendere l'esecuzione, si deve implementare del codice *ad hoc* che gestisca la situazione, dato che il linguaggio Java non mette a disposizione nessuno costrutto particolare.

Deamon thread

I thread in Java possono essere di due tipi: *user thread* o *deamon thread*. Il termine *deamon* è stato usato inizialmente per designare un certo tipo di processi nei sistemi operativi multitasking (in particolare in ambiente Unix), ossia dei processi "invisibili" che girano "in background" e svolgono dietro le quinte dei servizi di carattere generale. In genere questi processi restano in esecuzione per tutta la sessione del sistema. Il termine "demone" è stato usato probabilmente in analogia con "fantasma" a simboleggiare invisibilità e onnipresenza.

I deamon thread in Java sono qualcosa di molto simile ai processi deamon: sono infatti thread che spesso restano in esecuzione per tutta la durata di una sessione della virtual machine, ma soprattutto sono thread che si suppone che svolgano dei servizi per gli user thread, e che questa sia l'unica ragione della loro esistenza. In effetti l'unica differenza tra uno user thread e un deamon thread è che la virtual machine termina la sua esecuzione quando termina l'ultimo user thread, indipendentemente dal fatto che ci siano o meno in esecuzione dei deamon thread.

Spesso i deamon thread sono thread creati e mandati automaticamente in esecuzione dalla stessa virtual machine: un caso tipico è quello del garbage collector, che si occupa periodicamente di liberare la memoria allocata per oggetti non più in uso.

Ma un deamon thread può essere anche creato dall'utente, cioè dal programmatore: a tale scopo esiste il metodo `setDeamon(boolean value)` che permette di rendere deamon uno user thread o, viceversa, user un deamon thread.

Per default un thread, quando viene creato assume lo stato del thread "padre" da cui è stato creato. Con `setDeamon()` è possibile modificare questo stato, ma soltanto prima di mandare in esecuzione il thread con `start()`.

Una chiamata durante l'esecuzione causerà un'eccezione. Per conoscere il deamon state di un thread si può usare il metodo `isDeamon()`.

Se si creano dei thread di tipo deamon, occorre sempre tener presente che non devono svolgere delle operazioni che possano protrarsi oltre la durata di esecuzione degli user thread per cui svolgono i loro servizi. Questo rischierebbe di interrompere a metà queste operazioni, perché la Virtual Machine potrebbe terminare per mancanza di user thread in esecuzione.

I gruppi di thread

Ogni thread in Java appartiene a un gruppo; per default il gruppo di appartenenza è quello del thread padre. La virtual machine genera automaticamente dei thread groups, di cui almeno uno è destinato ai thread creati dalle applicazioni; questo sarà il gruppo di default per i thread creati da un'applicazione. Ogni applicazione può anche creare i suoi thread group, e assegnare i thread a questi gruppi.

I thread group sono organizzati secondo una struttura gerarchica ad albero: ciascun thread appartiene a un gruppo, il quale può appartenere a un altro gruppo; per questo ogni gruppo può contenere sia thread che gruppi di thread. La radice di quest'albero è rappresentata dal system thread group.

Per creare un thread group si deve creare un oggetto della classe `ThreadGroup` usando uno dei due costruttori:

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name);
```

Come per i thread, se non viene specificato un thread group di appartenenza, il gruppo di appartenenza sarà quello del thread da cui è stato creato.

Per assegnare un thread a un gruppo si usa uno dei tre costruttori:

```
Thread(ThreadGroup group, String name),
Thread(ThreadGroup group, Runnable target),
Thread(ThreadGroup group, Runnable target, String name).
```

Una volta creato il thread come membro di un certo gruppo, non è possibile farlo passare ad un altro gruppo o toglierlo dal gruppo. Il thread sarà rimosso automaticamente dal gruppo una volta terminata la sua esecuzione.

Le funzionalità relative ai gruppi di thread si possono suddividere in quattro categorie: funzionalità di informazione, manipolazione collettiva dei thread appartenenti a un gruppo, funzioni relative alla priorità e funzioni legate alla sicurezza. Nei paragrafi successivi si analizzano tali funzionalità.

Informazioni sui thread e sui gruppi

Ci sono diversi metodi appartenenti alla classe `ThreadGroup` e alla classe `Thread` che forniscono informazioni sui thread e sui gruppi di thread.

Ci sono metodi che ci informano su quanti e quali sono i thread e i gruppi attualmente esistenti nella VM.

Il più importante è il metodo `enumerate()`, che fornisce la lista dei thread o dei thread group attivi, effettuando opzionalmente una ricorsione in tutti i sottogruppi.

Vi sono poi metodi che informano sui "rapporti di parentela" come `getThreadGroup()` della classe `Thread` o `getParent()` della `ThreadGroup` che permettono di conoscere il gruppo di appartenenza di un thread o di un gruppo.

Thread group e priorità

Con il metodo `setMaxPriority(int` *`priorità`*`)` è possibile assegnare a un gruppo una priorità massima. Se si tenta di assegnare a un thread del gruppo (o di sottogruppi) una priorità maggiore, questa viene automaticamente ridotta alla priorità massima del gruppo, senza che venga segnalato alcun errore.

La priorità massima può essere soltanto diminuita, e non aumentata. La priorità dei thread appartenenti al gruppo non viene in realtà modificata se si abbassa la priorità massima del gruppo, anche se è più alta di tale limite. La limitazione diviene attiva solo quando viene creato un nuovo thread o viene modificata la priorità di un thread con il metodo `setPriority()`. In questo caso non sarà possibile superare la priorità massima del gruppo.

Il valore della priorità di un gruppo può essere ottenuto con una chiamata al metodo `getPriority()`.

Thread group e sicurezza

Le funzionalità più interessanti e più importanti legate ai thread group sono quelle relative alla sicurezza. Con i gruppi di thread è possibile consentire o interdire in maniera selettiva a interi gruppi di thread l'accesso ad altri thread e gruppi di thread.

Questa funzionalità è legata al funzionamento della classe `java.lang.SecurityManager`, la quale gestisce diverse funzioni legate alla sicurezza, tra cui alcune relative ai thread. In realtà queste funzioni sono riferite ai thread group, ma anche ai singoli thread.

Tuttavia i thread group assumono una particolare rilevanza perché consentono di discriminare l'accesso tra i thread sulla base dell'appartenenza ai gruppi, quindi accre-

scendo notevolmente le possibilità di organizzare i criteri di accesso secondo regole ben precise.

Il SecurityManager è quello che, ad esempio, si occupa di garantire che le Applet non possano accedere a determinate risorse del sistema. In questo caso si tratta di un SecurityManager fornito e gestito dal browser e dalla virtual machine del browser, a cui l'utente non ha accesso.

Ma per le applicazioni l'utente può invece creare e installare dei suoi SecurityManager. Prima di Java 2, le applicazioni non avevano nessun SecurityManager di default, quindi c'era solo la possibilità di usare dei SecurityManager creati dall'utente.

In Java 2 esiste anche un SecurityManager di default per le applicazioni, che può essere fatto partire con una opzione della VM, e configurato attraverso una serie di files di configurazione.

Tralasciando una descrizione complessiva del SecurityManager, le funzioni relative ai thread si basano in pratica su due soli metodi, o più precisamente su due versioni dello stesso metodo `checkAccess()`: questo infatti può prendere come parametro un oggetto `Thread` oppure un oggetto `ThreadGroup`.

A loro volta, le classi `Thread` e `ThreadGroup` contengono ciascuna un metodo `checkAccess()` che chiama i rispettivi metodi del SecurityManager.

Questo metodo viene chiamato da tutti i metodi della classe `Thread` e della classe `ThreadGroup` che determinano un qualsiasi cambiamento di stato nell'oggetto per cui vengono chiamati, per accertare che il thread corrente abbia il permesso di manipolare il thread in questione (che può essere lo stesso thread corrente o un altro thread).

Se le condizioni di accesso non sussistono, `checkAccess()` lancia una `SecurityException` che in genere viene semplicemente rilanciata dal metodo chiamante.

Le condizioni di accesso sono quindi stabilite dai metodi del SecurityManager, che può a tale scopo utilizzare tutte le informazioni che è in grado di conoscere sugli oggetti `Thread` o `ThreadGroup` di cui deve fare il check. Ad esempio può vietare l'accesso se il thread corrente e il thread in esame non appartengono allo stesso gruppo, o a seconda delle rispettive priorità, ecc.

I metodi che chiamano `checkAccess()` prima di compiere le loro operazioni sono:

nella classe `Thread`: `Thread()`, `interrupt()`, `setPriority()`, `setDeamon()`, `setName()`, più i metodi deprecati: `stop()`, `suspend()`, `resume()`;

nella classe `ThreadGroup`: `ThreadGroup()`, `interrupt()`, `setMaxPriority()`, `setDeamon()`, `destroy()`, più i metodi deprecati: `stop()`, `suspend()`, `resume()`.

Tutti questi metodi lanciano una `SecurityException` se il thread corrente non ha accesso al `Thread` o al `ThreadGroup` dell'oggetto `this`.

La classe `ThreadLocal`

Questa classe è stata introdotta con Java 2. Consente di avere una variabile locale al thread, cioè ciascun thread ha una sua istanza della variabile. Il valore della variabile è ottenuto tramite i metodi `get()` e `set()` dell'oggetto `ThreadLocal`. L'uso tipico di quest'oggetto è come variabile privata statica di una classe in cui si vuole mantenere uno stato o un identificatore per ogni thread.

Bibliografia

[tutor1] MARY CAMPIONE, KATHY WALRATH*The Java Tutorial Second Edition. Object-Oriented Programming for the Internet*. Ed. Addison-Wesley Pub Co. ISBN 0201310074
Disponibile anche in formato elettronico presso http://java.sun.com/docs/books/tutorial/

[thread]. SCOTT OAKS, HENRY WONG*The Java Thread (2a ed.)*. O'Reilly. ISBN 1-56592-418-5

[tij] BRUCE ECKEL,*Thinking in Java*. Ed. Prentice Hall. ISBN 0-13-659723-8

[java11] PHILIP HELLER, SIMON ROBERTS*Java 1.1 Developer's Handbook*. Ed. Sybex. ISBN 0-7821-1919-0

Input/Output

DI LORENZO BETTINI

Introduzione

In questo capitolo verrà illustrato il package `java.io`, che supporta il sistema fondamentale di input/output (I/O) di Java.

Nei programmi Java vengono spesso utilizzate istruzioni per stampare sullo schermo delle stringhe; utilizzare l'interfaccia a caratteri, invece che quella grafica, risulta molto comodo sia per scrivere esempi semplici, che per stampare informazioni di debug. Del resto se si scrive un'applicazione che utilizza intensamente la grafica, è comunque possibile stampare informazioni in una finestra di testo. In effetti il supporto di Java per l'I/O della console (testo) è un po' limitato, e presenta qualche complessità di utilizzo, anche nei programmi più semplici.

Comunque Java fornisce un ottimo supporto per l'I/O per quanto riguarda i file e la rete, tramite un sistema stabile e coerente. Si tratta di un ottimo esempio di libreria orientata agli oggetti che permette di sfruttare a pieno le feature della programmazione object oriented. Una volta compresi i concetti fondamentali dell'I/O di Java, è semplice sfruttare la parte restante del sistema I/O e, se si progettano le proprie classi tenendo presente la filosofia object oriented, si noterà come tali classi saranno riutilizzabili, ed indipendenti dal particolare mezzo di input/output.

Stream

I programmi in Java comunicano (cioè effettuano l'I/O) tramite gli *stream* (in italiano *flussi*). Uno stream è un'astrazione ad alto livello che produce o consuma informazioni: rappresenta una connessione a un canale di comunicazione. Uno stream quindi è collegato a un dispositivo fisico dal sistema I/O di Java. Gli stream possono sia leggere da un

canale di comunicazione che scrivere su tale canale: quindi si parla di stream di input, e stream di output.

Gli stream si comportano in modo omogeneo, indipendentemente dal dispositivo fisico con cui sono collegati (da qui il concetto di astrazione ad alto livello). Infatti le stesse classi e gli stessi metodi di I/O possono essere applicati a qualunque dispositivo. Uno stream (astratto) di input può essere utilizzato per leggere da un file su disco, da tastiera, o dalla rete; allo stesso modo uno stream di output può fare riferimento alla console (e quindi scrivere sullo standard output), a un file (e quindi scrivere e aggiornare un file), o ancora ad una connessione di rete (e quindi spedire dei dati in rete).

Un flusso quindi rappresenta un'estremità di un canale di comunicazione a un senso solo. Le classi di stream forniscono metodi per leggere da un canale o per scrivere su un canale. Quindi un *output stream* scrive dei dati su un canale di comunicazione, mentre un *input stream* legge dati da un canale di comunicazione. Non esistono delle classi di stream che forniscano funzioni sia per leggere che per scrivere su un canale. Se si desidera sia leggere che scrivere su uno stesso canale di comunicazione si dovranno aprire due stream (uno di input ed uno di output) collegati allo stesso canale.

Di solito un canale di comunicazione collega uno stream di output al corrispondente stream di input. Tutti i dati scritti sullo stream di output, potranno essere riletti (nello stesso ordine) dallo stream di input. Poiché, come si è già detto, gli stream sono indipendenti dal particolare canale di comunicazione, essi mettono a disposizione uno strumento semplice e uniforme per la comunicazione fra applicazioni. Due applicazioni che si trovano su due macchine diverse, ad esempio, potrebbero scambiarsi i dati tramite uno stream collegato alla rete, oppure un'applicazione può semplicemente comunicare con l'utente utilizzando gli stream collegati alla console. Gli stream implementano una struttura FIFO (*First In First Out*), nel senso che il primo dato che sarà scritto su uno stream di output sarà il primo che verrà letto dal corrispondente stream di input. Fondamentalmente, quindi, gli stream mettono a disposizione un accesso sequenziale alle informazioni scambiate.

Quando si parla di input/output, si parla anche del problema dell'azione bloccante di una richiesta di input (il concetto di input/output tra l'altro si ritrova anche nelle architetture dei processori). Ad esempio, se un thread cerca di leggere dei dati da uno stream di input che non contiene dati, verrà bloccato finché non saranno presenti dei dati disponibili per essere letti. In effetti, quando un thread cerca di leggere dei caratteri immessi da un utente da tastiera, rimarrà in attesa finché l'utente non inizierà a digitare qualcosa. Il problema dell'azione bloccante è valido anche per le operazioni di output: se si cerca di scrivere qualcosa in rete, si rimarrà bloccati finché l'operazione non sarà terminata. Questo può avvenire anche quando si scrive su un file su disco, ma le operazioni in rete di solito sono le più lente.

Il thread bloccato sarà risvegliato solo quando sarà stata completata l'operazione bloccante. Se si vuole evitare di essere bloccati da queste operazioni si dovrà utilizzare il multithreading; si vedranno degli esempi nel capitolo che riguarda il networking.

Le classi

Le classi degli stream sono contenute nel pacchetto `java.io`, che dovrà quindi essere incluso nei programmi che ne fanno uso.

Tutti gli stream fanno parte di una gerarchia. In realtà si hanno due sottogerarchie: una per gli stream di output ed una per quella di input.

In cima a questa gerarchia ci sono due classi astratte i cui nomi sono abbastanza ovvi: `InputStream` e `OutputStream`. Trattandosi di classi astratte, non si potranno istanziare direttamente oggetti appartenenti a queste classi. Comunque si possono dichiarare delle variabili appartenenti a queste classi (per i programmatori C++, si ricorda che le variabili dichiarate sono in effetti dei riferimenti o puntatori, e quindi dichiarando una variabile non si istanzia automaticamente un oggetto di tale classe), e a queste si potrà assegnare qualsiasi oggetto appartenente ad una classe derivata (l'analogia con il C++ prosegue: un puntatore a una classe base può puntare a un qualsiasi oggetto appartenente a una classe derivata); in questo modo si potrà utilizzare a pieno il polimorfismo, rendendo le proprie classi indipendenti dal particolare stream (e quindi anche dal particolare canale di comunicazione).

Java ovviamente mette a disposizione diverse sottoclassi che specializzano gli stream per i diversi dispositivi e canali di comunicazione, ma vediamo prima i metodi messi a disposizione da queste due classi base.

La classe `OutputStream`

La classe `OutputStream` rappresenta una porta verso un canale di comunicazione; tramite questa porta si possono scrivere dati sul canale con il quale la porta è collegata. Si ricorda che si tratta di una classe astratta, che quindi fornisce un'interfaccia coi metodi caratteristici di ogni stream di output. Saranno le sottoclassi a fornire un'implementazione effettiva di tali metodi, che ovviamente dipenderà dal particolare canale di comunicazione.

Descrizione classe

```
public abstract class OutputStream extends Object
```

Trattandosi di una classe astratta, non sono presenti costruttori utilizzabili direttamente.

Metodi

```
public abstract void write(int b) throws IOException
```

Viene accettato un singolo byte, che verrà scritto sul canale di comunicazione con il

quale lo stream è collegato. Notare che, nonostante l'argomento sia di tipo intero, verrà scritto solo il byte meno significativo. Ovviamente si tratta di un metodo astratto, in quanto la scrittura dipende fortemente dal particolare dispositivo fisico del canale di comunicazione.

```
public void write(byte b[], int off, int len) throws IOException
```

```
public void write(byte b[]) throws IOException
```

Questi metodi permettono di scrivere un array di byte sul canale di comunicazione. È possibile scrivere l'intero array (secondo metodo), o solo una parte (primo metodo), specificando l'indice del primo elemento (`off`), e il numero di elementi (`len`). Il secondo metodo, nell'implementazione di default, richiama semplicemente il primo sull'intero array. A sua volta il primo metodo, nella sua implementazione di default, richiama il numero di volte necessario il metodo `write(int b)`. Il metodo bloccherà il chiamante fino a che tutti i byte dell'array non saranno stati scritti.

```
public void flush() throws IOException
```

Questo metodo effettua il *flush* dei dati bufferizzati nello stream, cioè fa in modo che eventuali dati non ancora scritti effettivamente, vengano scritti nel canale di comunicazione. A volte infatti, per motivi di ottimizzazione e performance, i dati scritti nello stream non vengono scritti immediatamente nel canale di comunicazione, ma vengono tenuti temporaneamente in un buffer. Con questo metodo si fa in modo che i dati presenti nel buffer vengano scritti effettivamente sul canale. Quando si tratta di comunicazioni in rete, la tecnica della "bufferizzazione" è quasi d'obbligo, per ovviare alla lentezza di tali comunicazioni.

```
public void close() throws IOException
```

Con questo metodo si chiude lo stream e quindi il canale di comunicazione. Prima della chiusura tutti i dati eventualmente bufferizzati vengono sottoposti a flush; questo può comportare il dover attendere (e quindi rimanere bloccati) fino al completamente dell'operazione di scrittura.

L'eccezione `IOException` può essere lanciata per vari motivi che riguardano dei problemi del canale di comunicazione. Il tipo esatto dell'eccezione dipende quindi dal particolare canale. Tipicamente le operazioni sugli stream dovrebbero essere racchiuse nei classici blocchi `try-catch-finally`, o fare in modo che il metodo che li utilizza dichiari di lanciare una tale eccezione.

La classe `InputStream`

La classe `InputStream` è la classe complementare della classe `OutputStream`, che fornisce funzionalità per l'input, quindi per la lettura di dati da un canale di comunicazione. Quanto si è detto sui metodi astratti è valido anche per questa classe.

Descrizione classe

```
public abstract class InputStream extends Object
```

Metodi

Questa classe fornisce metodi per leggere byte, per determinare il numero di byte disponibili per essere letti senza rimanere bloccati, e per saltare o rileggere dei dati. Come è già stato accennato, leggere da uno stream che non contiene dati bloccherà il thread che ha effettuato l'operazione di lettura. Se alcuni dati sono già arrivati dal canale di comunicazione, verranno messi temporaneamente in un buffer in attesa di essere effettivamente letti. Quando, a questo punto, un thread cercherà di leggere dallo stream, lo potrà fare immediatamente senza bisogno di attendere e di bloccarsi.

```
public abstract int read() throws IOException
```

Questo metodo legge un singolo byte, aspettando eventualmente che ve ne sia uno disponibile. Ancora una volta, pur trattandosi di un `int`, il valore restituito sarà comunque compreso fra `0` e `255`. Se viene raggiunta la fine dello stream, verrà restituito il valore `-1`. Il concetto di fine dello stream dipende dal particolare canale di comunicazione che si sta utilizzando (ad esempio nel caso di un file rappresenta la fine del file). Si tratta di un metodo astratto perché la lettura di dati da uno stream dipende dal particolare canale di comunicazione con cui lo stream è collegato.

```
public int read(byte b[], int off, int len) throws IOException

public int read(byte b[]) throws IOException
```

Con questi metodi è possibile leggere una serie di byte e memorizzarli nell'array specificato. È possibile specificare anche il numero di byte da leggere (`len`) e memorizzare nell'array, specificando l'indice iniziale (`off`). L'array dovrà già essere stato allocato. Si tratta ovviamente di un metodo bloccante, se non sono presenti dati da leggere. Il metodo restituisce inoltre il numero di byte letti. Infatti non è detto che venga letto esattamente il numero di byte richiesti: vengono letti i dati che possono essere letti immediatamente

senza necessità di attendere, e questi possono essere in numero inferiore a quelli effettivamente richiesti. L'implementazione di default del secondo metodo è quella di richiamare il primo su tutto l'array. A sua volta l'implementazione di default del primo è di richiamare ripetutamente il metodo `read()`.

```
public abstract int available() throws IOException
```

Restituisce il numero di byte che sono disponibili nello stream per essere letti senza attendere.

```
public void close() throws IOException
```

Chiude lo stream e il canale di comunicazione con cui lo stream è collegato. I dati non ancora letti andranno persi.

```
public long skip(long n) throws IOException
```

Vengono saltati e scartati `n` byte presenti nello stream. Questo è utile se si vogliono ignorare dei byte, ed è più efficiente che leggere i byte e ignorarli. Il metodo restituisce il numero di byte effettivamente saltati; questo perché, per vari motivi, può non essere possibile saltare esattamente il numero di byte richiesto.

```
public synchronized void mark(int readlimit)
```

```
public synchronized void reset() throws IOException
```

Marca la posizione corrente all'interno dello stream. Una successiva chiamata al metodo `reset` riposizionerà lo stream alla precedente posizione marcata. Dopo la chiamata del metodo `reset` letture successive leggeranno dall'ultima posizione marcata. Con il parametro `readlimit` si specifica il numero massimo di byte che saranno letti, prima che la posizione marcata non sia più valida. Se sono letti più di `readlimit` byte, una successiva chiamata di `reset` potrebbe fallire.

Questi due metodi possono risultare utili nelle situazioni in cui vi sia bisogno di leggere alcuni byte prima di capire quale tipo di dati è presente nello stream. Se si deve decodificare tali dati, e si hanno vari tipi di decodificatori, quando un decodificatore si rende conto che non sono dati che lo riguardano, può "rimettere a posto" i dati già letti, rendendoli disponibili ad un altro decodificatore.

```
public boolean markSupported()
```

Permette di capire se lo stream corrente gestisce il corretto funzionamento delle operazioni di mark e reset.

Anche nel caso di `InputStream` l'eccezione `IOException` può essere lanciata in varie occasioni.

Gli stream predefiniti

Il pacchetto `java.lang`, incluso automaticamente da tutti i programmi Java, definisce alcuni stream predefiniti, contenuti nella classe `System`. Si tratta di tre variabili statiche e pubbliche (quindi utilizzabili in qualunque parte del programma, senza aver istanziato un oggetto `System`) denominate `in`, `out` e `err`. Queste si riferiscono rispettivamente allo standard input, che per default è la tastiera, al flusso standard di output, che per default è lo schermo, e al flusso standard di errori che, anche in questo caso, per default è lo schermo. Tali stream possono essere reindirizzati quando si lancia il programma da linea di comando utilizzando > e < (per questo si rimanda al sistema operativo che si utilizza).

Esempi

Si prenderanno ora in considerazione due semplici esempi che utilizzano tali stream predefiniti:

```
import  java.io.*;

public class OutSample {
   public static void main (String args[]) throws IOException {
      for (int i = 0; i < args.length; ++ i) {
         synchronized(System.out) {
            for (int j = 0; j < args[i].length (); ++j)
               System.out.write ((byte) args[i].charAt (j));
            System.out.write ('\n');   // scrive un invio
            System.out.flush ();    // scarica il buffer
         }
      }
   }
}
```

Questo semplice programma scrive sullo schermo i vari argomenti passati sulla linea di comando. Viene utilizzato il metodo `write` per scrivere un byte alla volta, ed il metodo `flush` per essere sicuri che ogni stringa passata venga stampata subito. Si può notare che il metodo `main` dichiara di poter lanciare un'eccezione `IOException`; in effetti i metodi `write` e `flush` possono lanciare tali eccezioni.

Un po' meno chiaro può risultare l'utilizzo di un blocco sincronizzato. In questo caso non sarebbe necessario in quanto non si usano più thread. Nel caso di un programma con

più thread è bene sincronizzare l'accesso alla variabile `System.out` in modo che, quando un thread ha iniziato a scrivere su tale stream, non venga interrotto prima che abbia finito; nello stream altrimenti sarebbero presenti informazioni rovinate e mischiate.

Un'alternativa potrebbe essere quella di scrivere una stringa alla volta, invece dei suoi singoli byte. Per far questo si deve convertire la stringa in un array di byte, e poi richiamare il metodo `write` appropriato. Vale a dire che al posto del ciclo `for` più interno si sarebbe potuto scrivere

```
byte buffer[] = new byte[args[i].length()];
msg.getBytes (0, args[i].length (), buffer, 0);
System.out.write (buffer);
```

In effetti la variabile `out` appartiene alla classe `PrintStream`, che specializza un `OutputStream` per scrivere dati in formato testo (e quindi adatto per scrivere dati sullo schermo). Questa classe mette a disposizione due metodi molto utilizzati per stampare facilmente stringhe e altri dati come testo: si tratta dei metodi `print` e `println` (quest'ultimo si distingue dal precedente perché aggiunge un `newline` dopo la stampa). In effetti il programma precedente può essere riscritto in modo molto più semplice:

```
public class OutSamplePrint {
   public static void main (String args[]) throws IOException {
      for (int i = 0; i < args.length; ++ i)
         System.out.println(i + ": " + args[i]);
   }
}
```

Come si vede non c'è bisogno di tradurre la stringa in un array di byte, in quanto i metodi suddetti gestiscono direttamente le stringhe, e sono anche in grado di tradurre dati di altro tipo (ad esempio `i` è un intero) in stringa (infatti questo programma stampa le stringhe immesse da riga di comando insieme alla numerazione). Non c'è nemmeno bisogno di sincronizzarsi su `System.out` in quanto questi metodi sono già dichiarati come sincronizzati.

Ecco adesso un semplice programma che legge i caratteri immessi da tastiera e li ristampa sullo schermo. In questo caso si utilizzerà anche la variabile `System.in`.

```
import  java.io.*;

public class InSample {
   static public void main (String args[]) throws IOException {
      int c;
      while ((c = System.in.read ()) >= 0)
         System.out.print((char)c);
```

```
   }
}
```

Come si vede, dei caratteri da tastiera vengono letti e poi stampati sullo schermo (la conversione esplicita a `char` è necessaria, altrimenti verrebbe stampato un numero). Da notare che il metodo `read` memorizza in un buffer i caratteri digitati e li restituisce solo quando l'utente preme `Invio`. Chiaramente questo metodo non è molto indicato per un input interattivo da console.

Ancora una volta, può essere più efficiente utilizzare dei buffer per ottimizzare le prestazioni. Per far questo basta cambiare il corpo del `main` con il seguente:

```
byte buffer[] = new byte[8];
int numberRead;
while ((numberRead = System.in.read (buffer)) > -1)
   System.out.write (buffer, 0, numberRead);
```

Questo semplice programma può essere utilizzato anche per visualizzare il contenuto di un file di testo: basterà semplicemente ridirezionare lo standard input (la tastiera) su un file. Ad esempio con il seguente comando

```
java InSample < InSample.java
```

si visualizzerà sullo schermo il contenuto del sorgente del programma stesso.

Si vedranno adesso alcune classi che specializzano gli stream di input e output. Come si è visto le classi base offrono solo metodi per scrivere singoli byte e al massimo array di byte. Spesso invece si ha la necessità di leggere e/o scrivere stringhe o numeri, quindi si avrebbe bisogno di stream che forniscano metodi per effettuare direttamente queste operazioni, senza dover manualmente effettuare conversioni.

Stream filtro

Si vedrà ora il concetto di stream filtro (*filter stream*), cioè uno stream che fornisce metodi ad alto livello per inviare o ricevere i dati primitivi di Java su un qualsiasi stream di comunicazione.

Uno stream filtro agisce appunto come un filtro per uno stream già esistente, aggiungendo funzionalità ad alto livello. Tra l'altro questo permette di tralasciare tutti i dettagli su come i dati vengono memorizzati in uno stream (ad esempio se un intero viene memorizzato partendo dal byte più alto o da quello più basso).

È necessario quindi fornire uno stream già esistente ad uno stream filtro. Ad uno stream filtro di input passeremo uno stream di input qualsiasi (in pratica un oggetto di classe

`InputStream`), così come ad uno stream filtro di output passeremo uno stream di output qualsiasi (un oggetto di classe `OutputStream`). Anche gli stream filtro sono sottoclassi delle classi base `InputStream` e `OutputStream`, quindi è possibile costruire una serie di stream filtro in cascata, a seconda delle varie esigenze. Ci sarà modo di vedere alcuni esempi successivamente.

Le classi `FilterOutputStream` e `FilterInputStream`

Queste sono le classi base per ogni stream filtro, e non sono altro che template (modelli) per tutti gli altri stream filtro. L'unica funzionalità aggiuntiva che mettono a disposizione è il fatto di poter passare ai loro costruttori un qualsiasi stream con il quale collegarsi (quindi rispettivamente un `OutputStream` e un `InputStream`). Gli unici metodi che mettono a disposizione sono gli stessi che sono presenti nella classe base. La semplice azione di default sarà quella di richiamare il metodo corrispondente dello stream con il quale sono collegati. La loro utilità si riduce quindi a fornire un'interfaccia uniforme per tutti gli altri stream filtro, e ovviamente a fornire una classe base comune.

Le classi `DataOutputStream` e `DataInputStream`

Queste classi sono fra le più utilizzate in quanto mettono a disposizione proprio le funzionalità che cercavamo negli stream filtro: forniscono metodi rispettivamente per scrivere e leggere tutti i tipi primitivi del linguaggio (stringhe, interi, ecc.).

Ovviamente questi due stream, come spesso accade negli stream filtro, devono lavorare in coppia affinché le comunicazioni di informazioni abbiano successo: se da una parte si utilizza un `DataOutputStream` per spedire una stringa, dall'altra parte ci dovrà essere in ascolto un `DataInputStream`, che sia in grado di decodificare la stringa ricevuta dal canale di comunicazione. Infatti i metodi di questi stream filtro si occupano, rispettivamente, di codificare e decodificare i vari tipi di dato. Non sarà necessario preoccuparsi dell'ordine dei byte di un intero o della codifica di una stringa, ovviamente purché tali stream siano utilizzati in coppia.

Nonostante non ci si debba preoccupare della codifica dei dati spediti, può comunque essere interessante sapere che questi stream utilizzano il *network byte order* per la memorizzazione dei dati: il byte più significativo viene scritto per primo (e dall'altra parte letto per primo). In questo modo le applicazioni scritte in Java, potranno comunicare dati con questi stream, con qualsiasi altro programma scritto in un altro linguaggio che usi la convenzione del network byte order.

Descrizione classe `DataOutputStream`

```
public class DataOutputStream
```

```
extends FilterOutputStream implements DataOutput
```

L'unica cosa da notare è l'interfaccia `DataOutput`. Questa interfaccia, insieme alla simmetrica `DataInput`, descrive gli stream che scrivono e leggono (rispettivamente) dati in un formato indipendente dalla macchina.

Costruttore

```
public DataOutputStream(OutputStream out)
```

Come già accennato quando si è parlato in generale degli stream filtro, viene passato al costruttore lo stream sul quale si agisce da filtro. Vale la pena di ricordare che si passa un `OutputStream`, quindi, trattandosi della classe base di tutti gli stream di output, si può passare un qualsiasi stream di output.

Metodi

I metodi seguenti fanno parte della suddetta interfaccia `DataOutput`. A questi vanno aggiunti i metodi derivati dalla classe base, che non verranno descritti (il loro nome è di per sé molto esplicativo).

```
public final void writeBoolean(boolean v) throws IOException

public final void writeByte(int v) throws IOException

public final void writeShort(int v) throws IOException

public final void writeChar(int v) throws IOException

public final void writeInt(int v) throws IOException

public final void writeLong(long v) throws IOException

public final void writeFloat(float v) throws IOException

public final void writeDouble(double v) throws IOException
```

Come si può notare, esiste un metodo per ogni tipo di dato primitivo di Java. Il loro significato dovrebbe essere abbastanza immediato. I prossimi metodi invece meritano una spiegazione un po' più dettagliata.

```
public final void writeBytes(String s) throws IOException
```

Questo metodo scrive una stringa sullo stream collegato come una sequenza di byte. Viene scritto solo il byte più basso di ogni carattere, quindi può essere utilizzato per trasmettere dei dati in formato ASCII a un dispositivo come un terminale carattere, o un client scritto in C. La lunghezza della stringa non viene scritta nello stream.

```
public final void writeChars(String s) throws IOException
```

La stringa passata viene scritta come sequenza di caratteri. Ogni carattere viene scritto come una coppia di byte. Non viene scritta la lunghezza della stringa, né il terminatore.

```
public final void writeUTF(String str) throws IOException
```

La stringa viene scritta nel formato Unicode UTF-8 in modo indipendente dalla macchina. La stringa viene scritta con una codifica in modo tale che ogni carattere viene scritto come un solo byte, due byte, o tre byte. I caratteri ASCII saranno scritti come singoli byte, mentre i caratteri più rari vengono scritti con tre byte. Inoltre i primi due byte scritti rappresentano il numero di byte effettivamente scritti. Quindi la lunghezza della stringa viene scritta nello stream.

Tutti questi metodi possono lanciare l'eccezione `IOException`; questo perché viene usato il metodo `write` dello stream con il quale lo stream filtro è collegato, che può lanciare un'eccezione di questo tipo.

Descrizione classe `DataInputStream`

```
public class DataInputStream extends FilterInputStream implements DataInput
```

Valgono le stesse considerazioni fatte riguardo alla classe `DataOutputStream`.

Costruttore

```
public DataInputStream(InputStream in)
```

Anche in questo caso si passa un `InputStream` al costruttore.

Metodi

Sono presenti i metodi simmetrici rispetto a quelli di `DataOutputStream`.

```
public final boolean readBoolean() throws IOException

public final byte readByte() throws IOException

public final int readUnsignedByte() throws IOException

public final short readShort() throws IOException

public final int readUnsignedShort() throws IOException

public final char readChar() throws IOException

public final int readInt() throws IOException

public final long readLong() throws IOException

public final float readFloat() throws IOException

public final double readDouble() throws IOException

public final String readUTF() throws IOException
```

Metodi che meritano particolare attenzione sono i seguenti:

```
public final void readFully(byte b[], int off, int len) throws IOException

public final void readFully(byte b[]) throws IOException
```

Questi metodi leggono un array di byte o un sottoinsieme, ma bloccano il thread corrente finché tutto l'array (o la parte di array richiesta) non viene letto. Viene lanciata un'eccezione `EOFException` se viene raggiunto prima l'EOF. A tal proposito si può notare che non può essere restituito il numero `-1` per segnalare l'EOF, in quanto se si sta leggendo un intero, `-1` è un carattere intero accettabile. Per questo motivo si ricorre all'eccezione suddetta.

```
public final static String readUTF(DataInput in) throws IOException
```

Si tratta di un metodo statico che permette di leggere una stringa con codifica UTF, dall'oggetto `in`, quindi un oggetto (in particolare uno stream) che implementi l'interfaccia `DataInput`.

Anche in questo caso può essere lanciata un'eccezione `IOException`. In particolare la suddetta eccezione `EOFException` deriva da `IOException`. Un'altra eccezione (sempre derivata da `IOException`) che può essere lanciata è `UTFDataFormatException`, nel caso in cui i dati ricevuti dal metodo `readUTF` non siano nel formato UTF.

Classi `BufferedOutputStream` e `BufferedInputStream`

Talvolta nelle comunicazioni è molto più efficiente bufferizzare i dati spediti. Questo è senz'altro vero per le comunicazioni in rete, ma può essere vero anche quando si deve scrivere o leggere da un file (anche se a questo pensa automaticamente il sistema operativo sottostante).

Richiamando i metodi di scrittura della classe `BufferedOutputStream`, i dati verranno memorizzati temporaneamente in un buffer interno (quindi in memoria), finché non viene chiamato il metodo `flush`, che provvederà a scrivere effettivamente i dati nello stream con cui il filtro è collegato, oppure finché il buffer non diventa pieno.

Quindi è molto più efficiente scrivere dei dati su un canale di comunicazione utilizzando un `DataOutputStream` collegato a un `BufferedOutputStream`. Ad esempio se si utilizza un `DataOutputStream` collegato direttamente a un canale di comunicazione di rete, e si scrive un intero con il metodo `writeInt`, è molto probabile che il primo byte dell'intero scritto sarà spedito subito in rete in un pacchetto. Un altro pacchetto — o forse più pacchetti — sarà utilizzato per i rimanenti byte. Sarebbe molto più efficiente scrivere tutti i byte dell'intero in un solo pacchetto e spedire quel singolo pacchetto. Se si costruisce un `DataOutputStream` su un `BufferedOutputStream` si otterranno migliori prestazioni. Questo è, tra l'altro, un esempio di due stream filtro collegati in cascata.

Se da una parte della comunicazione c'è un `BufferedOutputStream` che scrive dei dati, non è detto che dall'altra parte ci debba essere un `BufferedInputStream` in ascolto: in effetti questi stream filtro non codificano l'output ma semplicemente effettuano una bufferizzazione.

Comunque converrebbe utilizzare anche in lettura uno stream bufferizzato, cioè un `BufferedInputStream`. Utilizzare un buffer in lettura significa leggere i dati dal buffer interno, e solo quando nuovi dati, non presenti nel buffer, dovranno essere letti, si accederà al canale di comunicazione.

Descrizione classe `BufferedOutputStream`

```
public  class  BufferedOutputStream
extends FilterOutputStream
```

Costruttori

```
public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int size)
```

Nel primo caso viene creato uno stream bufferizzato collegato allo stream di output `out`; la dimensione del buffer sarà quella di default, cioè 512 byte. Nel secondo caso è

possibile specificare la dimensione del buffer. I dati scritti in questo stream saranno scritti sullo stream collegato `out` solo quando il buffer è pieno, o verrà richiamato il metodo `flush`.

Metodi

Come si è visto la classe deriva direttamente da `FilterOutputStream` e l'unico metodo che aggiunge a quelli della classe base (cioè quelli di `OutputStream`) è il metodo `flush`.

```
public synchronized void flush() throws IOException
```

Questo metodo fa sì che i dati contenuti nel buffer siano effettivamente scritti sullo stream collegato.

Descrizione classe `BufferedInputStream`

```
public class BufferedInputStream extends FilterInputStream
```

Costruttori

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
```

Viene creato uno stream di input bufferizzato collegato allo stream si input `in`; è possibile anche in questo caso specificare la dimensione del buffer, o accettare la dimensione di default (512 byte).

Metodi

Non vengono aggiunti metodi, e quindi si hanno a disposizione solo quelli di `FilterInputStream` (cioè solo quelli di `InputStream`); l'unica differenza è che tali metodi faranno uso del buffer interno.

Stream per l'accesso alla memoria

Java supporta l'input e l'output di array di byte tramite l'uso delle classi `ByteArrayOutputStream` e `ByteArrayInputStream`. Questi stream non sono collegati con un canale di comunicazione vero e proprio: queste classi infatti utilizzano dei

buffer di memoria come sorgente e come destinazione dei flussi di input e output. In questo caso, tali stream non devono essere utilizzati necessariamente insieme.

Vi sono poi le classi `PipedInputStream` e `PipedOutputStream`, che permettono la comunicazione, tramite appunto memoria, di due thread di un'applicazione. Un thread leggerà da un lato della *pipe* e riceverà tutto quello che sarà scritto dall'altro lato da altri thread. Questi stream saranno creati sempre in coppia; un lato della *pipe* viene creato senza essere connesso, mentre l'altro sarà creato connettendolo con il primo. Quindi basta collegare uno stream con l'altro, e non entrambi.

Descrizione classe `ByteArrayInputStream`

```
public class ByteArrayInputStream extends InputStream
```

Questa classe crea uno stream di input da un buffer di memoria, in particolare da un array di byte.

Costruttori

```
public ByteArrayInputStream(byte buf[])
public ByteArrayInputStream(byte buf[], int offset, int length)
```

Con questi costruttori si può specificare l'array (o una parte dell'array nel secondo caso) con il quale lo stream sarà collegato.

Metodi

Tale classe non mette a disposizione nuovi metodi, semplicemente ridefinisce i metodi della classe base `InputStream`. In particolare chiamando il metodo `read`, in una delle sue forme, verranno letti i byte dell'array collegato, fino a che non sarà raggiunta la fine dell'array, e in tal caso sarà restituito `EOF`. Inoltre la semantica del metodo `reset` è leggermente differente: resettare un `ByteArrayInputStream` vuol dire ripartire sempre dall'inizio dell'array, in quanto il metodo `mark` marca sempre la posizione iniziale.

Descrizione classe `ByteArrayOutputStream`

```
public class ByteArrayOutputStream extends OutputStream
```

Questa classe crea uno stream di output su un array di byte, ed è un po' più potente della sua classe complementare: permette all'array di byte con il quale è collegata di cre-

scere dinamicamente, man mano che vengono aggiunti nuovi dati. Il buffer di memoria può essere estratto e utilizzato.

Costruttori

```
public ByteArrayOutputStream()
```

```
public ByteArrayOutputStream(int size)
```

È possibile specificare la dimensione iniziale del buffer o accettare quella di default (32 byte).

Metodi

Anche in questo caso il metodo `reset()` acquista un significato particolare: svuota il buffer, e successive scritture memorizzeranno i dati a partire dall'inizio. Vi sono poi alcuni metodi aggiunti:

```
public int size()
```

Viene restituito il numero di byte che sono stati scritti nel buffer (da non confondersi con la dimensione del buffer, che può essere anche maggiore).

```
public synchronized byte[] toByteArray()
```

Viene restituito un array di byte rappresentante una copia dei dati scritti nel buffer. Il buffer interno non sarà resettato da questo metodo, quindi successive scritture nello stream continueranno a estendere il buffer.

```
public String toString()
```

Viene restituita una stringa rappresentante una copia del buffer dello stream. Anche in questo caso il buffer dello stream non viene resettato. Ogni carattere della stringa corrisponderà al relativo byte del buffer.

```
public synchronized void writeTo(OutputStream out) throws IOException
```

I contenuti del buffer dello stream vengono scritti nello stream di output `out`. Anche in questo caso il buffer dello stream non viene resettato. Se si verificano degli errori durante la scrittura nello stream di output `out` verrà sollevata un'eccezione `IOException`.

Ecco adesso un piccolo esempio che fa uso dei suddetti stream. Le stringhe che vengono passate sulla riga di comando vengono tutte inserite in `ByteArrayOutputStream`. Il buffer dello stream viene estratto e su tale array di byte viene costruito un `ByteArrayInputStream`. Da questo stream verranno poi estratti e stampati sullo schermo tutti i byte.

```
Import java.io.* ;

public class ByteArrayIOSample {
   public static void main(String args[]) throws IOException {
      ByteArrayOutputStream oStream = new ByteArrayOutputStream();

      for (int i = 0; i < args.length; i++)
         for (int j = 0; j < args[i].length(); j++)
            oStream.write(args[i].charAt(j));

            // per la concatenazione a stringa viene
            // chiamato toString()
      System.out.println("oStream: " + oStream);

      System.out.println("size: " + oStream.size());


ByteArrayInputStream iStream = new ByteArrayInputStream(oStream.toByteArray());

      System.out.println("Byte disponibili: " + iStream.available());
      int c ;
      while((c = iStream.read()) != -1)
         System.out.write(c);
   }
}
```

Descrizione classe `PipedOutputStream`

```
public  class  PipedOutputStream  extends  OutputStream
```

Costruttori

```
public  PipedOutputStream()  throws  IOException

public PipedOutputStream(PipedInputStream snk) throws IOException
```

Si può creare un `PipedOutputStream` e poi connetterlo con un `PipedInputStream`, oppure lo si può passare direttamente al costruttore, se già esiste.

Metodi

Sono disponibili i metodi standard della classe `OutputStream` ed in più è presente il metodo per connettere lo stream con un `PipedInputStream`:

```
public void connect(PipedInputStream src) throws IOException
```

Se si scrive su un `PipedOutputStream`, e il thread che è in ascolto sul corrispondente `PipedInputStream` termina, si otterrà un'`IOException`.

Questi stream sono implementati con un buffer di memoria, e se il buffer diventa pieno, una successiva chiamata al metodo `write` bloccherà il thread che scrive sullo stream, finché il thread in ascolto sullo stream di input corrispondente non legge qualche byte. Se questo thread termina, l'eccezione suddetta evita che l'altro processo rimanga bloccato indefinitamente.

Descrizione classe `PipedInputStream`

Per questa classe, che è la relativa classe di lettura della precedente, valgono le stesse annotazioni fatte per la classe `PipedOutputStream`.

```
public class PipedInputStream extends InputStream
```

Costruttori

```
public PipedInputStream() throws IOException

public PipedInputStream(PipedOutputStream src) throws IOException
```

Metodi

```
public void connect(PipedOutputStream src) throws IOException
```

Anche in questo caso si deve evitare che un thread bloccato a leggere da un `PipedInputStream`, rimanga bloccato indefinitamente; se viene chiamato il metodo `read` su uno stream vuoto, verrà sollevata un'eccezione `IOException`.

Segue un semplice esempio che illustra l'utilizzo di questi due stream per la comunicazione fra due thread (il thread principale e un thread parallelo):

```
import java.io.* ;

public class PipedIOSample extends Thread {
   protected DataInputStream iStream ;
```

```
   public  PipedIOSample(InputStream  i)  {
         this.iStream = new DataInputStream(i);
   }

   public void run() {
      try {
         String str;
         while (true) {
            str = iStream.readUTF();
            System.out.println("Letta: " + str);
         }
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }

   public static void main(String args[]) throws IOException {
      PipedOutputStream o = new PipedOutputStream();
      PipedInputStream iStream = new PipedInputStream(o);
      DataOutputStream oStream = new DataOutputStream(o);

      (new PipedIOSample(iStream)).start();

      for (int i = 0; i < args.length; i++) {
         System.out.println("Scrivo: " + args[i]);
         oStream.writeUTF(args[i]);
      }

      oStream.close();
   }
}
```

Come si può notare viene creato un `PipedOutputStream` senza specificare nessuno stream da collegare; poi viene creato un `PipedInputStream` collegato al precedente stream. A questo punto i due stream sono connessi e tutto quello che viene scritto sullo stream di output potrà essere letto da quello di input. L'idea è quella di scrivere le stringhe da passare sulla riga di comando sullo stream di output; tali stringhe saranno lette da un altro thread sullo stream (sempre di tipo piped) di input. In particolare invece di utilizzare più volte il metodo `write` per scrivere un singolo byte alla volta, utilizziamo un `DataOutputStream` collegato al `PipedOutputStream`, e scriviamo una stringa alla volta con il metodo `readUTF`. Allo stesso modo il thread che legge le stringhe lo farà tramite un `DataInputStream` collegato allo stream passato al costruttore. Vale la pena di notare che al costruttore viene passato un `InputStream` generico. Il thread che legge le stringhe lo fa in un ciclo infinito, che terminerà non appena verrà chiuso lo stream di output (ultima istruzione del main), a causa dell'eccezione `IOException`.

I file

Trattando l'input/output non si può certo tralasciare l'argomento file. Java fornisce l'accesso ai file tramite gli stream. In questo modo, per la genericità degli stream, un'applicazione progettata per leggere e/o scrivere utilizzando le classi `InputStream` e `OutputStream`, può utilizzare i file in modo trasparente.

Java inoltre mette a disposizione altre classi per facilitare l'accesso ai file e alle directory

Descrizione classe `File`

```
public class File extends Object implements Serializable
```

La classe `File` fornisce l'accesso a file e directory in modo indipendente dal sistema operativo. Tale classe mette a disposizione una serie di metodi per ottenere informazioni su un certo file e per modificarne gli attributi; tramite questi metodi, ad esempio, è possibile sapere se un certo file è presente in una certa directory, se è a sola lettura, e via dicendo.

Si è parlato di indipendenza dal sistema operativo: effettivamente ogni sistema operativo utilizza convenzioni diverse per separare le varie directory in un *path*. Quando si specifica un file e/o un path, si suppone che vengano utilizzate le convenzioni del sistema operativo sottostante. I vari metodi che sono messi a disposizione dalla classe permettono di ottenere le informazioni relative a tali convenzioni. Inoltre è possibile cancellare file, rinominarli, e ottenere la lista dei file contenuti in una certa directory.

Costruttori

```
public File(String path)
public File(String path, String name)
public File(File dir, String name)
```

È possibile creare un oggetto `File` specificando un `path` e anche un nome di file. Il `path` deve essere specificato utilizzando le convenzioni del sistema operativo sottostante. Se viene specificato anche il nome del file, oltre al percorso, verrà creato un `path` concatenando il percorso specificato ed il file con il separatore utilizzato dal sistema operativo. Con la terza versione è possibile specificare la directory del file tramite un altro oggetto `File`.

Metodi

Come già detto, tale classe è utile per avere un meccanismo in grado di utilizzare file e

directory in modo indipendente dalle convenzioni del sistema operativo e per eseguire le classiche operazioni sui file e sulle directory. Tali metodi non lanciano un'`IOException`, in caso di fallimento, ma restituiscono un valore booleano.

```
public String getName()

public String getPath()
```

Restituiscono rispettivamente il nome e il percorso dell'oggetto `File`.

```
public String getAbsolutePath()

public String getCanonicalPath() throws IOException
```

Restituiscono rispettivamente il percorso assoluto dell'oggetto `File`, e il percorso canonico. Quest'ultimo è un percorso completo in cui eventuali riferimenti relativi e simbolici sono già stati valutati e risolti. Quest'ultimo concetto ovviamente dipende fortemente dal sistema operativo.

```
public String getParent()
```

Restituisce il nome della *parent directory* dell'oggetto `File`. Per un file si tratta del nome della directory.

```
public boolean exists()

public boolean canWrite()

public boolean canRead()
```

Questi metodi permettono di capire se un file con il nome specificato esiste, se è scrivibile e se è leggibile.

```
public boolean isFile()

public boolean isDirectory()
```

```
public boolean isAbsolute()
```

Permettono di capire se l'oggetto `File` rappresenta un file, una directory o un percorso assoluto.

```
public long lastModified()
public long length()
```

Permettono di conoscere la data dell'ultima modifica del file, e la sua lunghezza in byte.

```
public boolean renameTo(File dest)
public boolean delete()
```

Permettono di rinominare e di cancellare un file.

```
public boolean mkdir()
public boolean mkdirs()
```

Permette di creare una directory che corrisponde all'oggetto `File`. La seconda versione crea se necessario tutte le directory del percorso dell'oggetto `File`.

```
public String[] list()
public String[] list(FilenameFilter filter)
```

Restituiscono l'elenco di tutti i file della directory corrispondente all'oggetto `File`. Nella seconda versione è possibile specificare un filtro.

Descrizione classe `RandomAccessFile`

```
public class RandomAccessFile extends Object implements DataOutput,
                                                        DataInput
```

Anche in Java è possibile accedere ai file in modo random, cioè non in modo sequenziale. Tramite questa classe infatti è possibile accedere a una particolare posizione in un file, ed è inoltre possibile accedere al file contemporaneamente in lettura e scrittura (cosa che

non è possibile con l'accesso sequenziale messo a disposizione dagli stream sui file, che saranno illustrati successivamente). È comunque possibile specificare in che modo accedere a un file (solo lettura, o lettura e scrittura).

La classe `RandomAccessFile`, implementando le interfacce `DataInput` e `DataOutput`, rende possibile scrivere in un file tutti gli oggetti e i tipi di dati primitivi. La classe inoltre fornisce i metodi per gestire la posizione corrente all'interno del file.

Se si scrive su un file esistente ad una particolare posizione si sovrascriveranno i dati a quella posizione.

Costruttori

```
public RandomAccessFile(String file, String mode) throws IOException

public RandomAccessFile(File file, String mode) throws IOException
```

Si può specificare il file da aprire sia tramite una stringa, sia tramite un oggetto `File`. Si deve inoltre specificare il modo di apertura del file nella stringa `mode`. Con la stringa `"r"` si apre il file in sola lettura, e con `"rw"` sia in lettura che in scrittura.

Metodi

```
public int read() throws IOException
```

Legge un byte. Blocca il processo chiamante se non è disponibile dell'input.

```
public int read(byte b[], int off, int len) throws IOException

public int read(byte b[]) throws IOException
```

Riempie un array o una parte dell'array specificato con i dati letti dal file. Viene restituito il numero di byte effettivamente letti.

```
public final void readFully(byte b[]) throws IOException

public final void readFully(byte b[], int off, int len) throws IOException
```

Questi metodi cercano di riempire un array (o una sua parte) con i dati letti dal file. Se viene raggiunta la fine del file prima di aver terminato, viene lanciata un'eccezione `EOFException`.

```
public final FileDescriptor getFD() throws IOException
```

Viene restituito un descrittore di file utilizzato dal sistema operativo per gestire il file. Si tratta di un descrittore a basso livello rappresentato dalla classe `FileDescriptor`. Difficilmente ci sarà la necessità di gestire direttamente tale informazione.

```
public int skipBytes(int n) throws IOException
```

Questo metodo salta `n` byte, bloccandosi finché non sono stati saltati. Se prima di questo si incontra la fine del file, viene sollevata un'eccezione `EOFException`.

```
public void write(int b) throws IOException
public void write(byte b[]) throws IOException
public void write(byte b[], int off, int len) throws IOException
```

Questi metodi permettono di scrivere rispettivamente in un file un singolo byte (nonostante l'argomento sia di tipo intero, solo il byte meno significativo viene effettivamente scritto nel file), un intero array, o una parte.

```
public native long getFilePointer() throws IOException
```

Restituisce la posizione corrente all'interno del file, cioè la posizione in cui si sta leggendo o scrivendo.

```
public void seek(long pos) throws IOException
```

Sposta il puntatore all'interno del file alla posizione assoluta specificata in `pos`.

```
public long length() throws IOException
```

Restituisce la lunghezza del file.

```
public void close() throws IOException
```

Chiude il file (scrivendo sul disco eventuali dati bufferizzati).

Nella classe sono poi presenti diversi metodi per leggere e scrivere particolari tipi di dati (ad esempio `readBoolean`, `writeBoolean`, `readInt`, `writeInt`, ecc.), come quelli già visti nelle classi `DataInputStream` e `DataOutputStream`, del resto, come abbiamo visto, `RandomAccessFile` implementa le interfacce `DataInput` e `DataOutput`. Per una lista completa si faccia riferimento alla guida in linea.

Le classi `FileOutputStream` e `FileInputStream`

Tramite queste classi è possibile accedere, rispettivamente in scrittura ed in lettura, sequenzialmente ai file, con il meccanismo degli stream.

Descrizione classe `FileOutputStream`

```
public class FileOutputStream extends OutputStream
```

Costruttori

```
public FileOutputStream(String name) throws IOException
public FileOutputStream(String name, boolean append) throws IOException
```

Si può aprire un file in scrittura specificandone il nome tramite una stringa. Se esiste già un file con lo stesso nome, verrà sovrascritto. È possibile (con il secondo costruttore) specificare se il file deve essere aperto in *append mode*.

```
public FileOutputStream(File file) throws IOException
```

Si può specificare il file da aprire tramite un oggetto `File` già esistente. Anche in questo caso, se il file esiste già, viene sovrascritto.

```
public FileOutputStream(FileDescriptor fdObj)
```

Si può infine specificare il file con cui collegare lo stream tramite un `FileDescriptor`. In questo modo si apre uno stream su un file già aperto, ad esempio uno aperto per accesso random. Ovviamente utilizzando questo costruttore non si crea (e quindi non si sovrascrive) un file, che anzi, come già detto, deve essere già aperto.

Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il `FileDescriptor` relativo al file collegato allo stream.

```
public native void close() throws IOException
```

Si dovrebbe chiamare sempre questo metodo quando non si deve più scrivere sul file. Questo metodo sarà comunque chiamato automaticamente quando lo stream sarà sottoposto al *garbage collecting*.

Descrizione classe `FileInputStream`

```
public  class  FileInputStream  extends  InputStream
```

Costruttori

```
public FileInputStream(String name) throws FileNotFoundException
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(FileDescriptor fdObj)
```

Uno stream può essere aperto specificando il file da aprire negli stessi modi visti nella classe `FileOutputStream`. Nel terzo caso il file è già aperto, ma nei primi due no: in tal caso il file deve esistere, altrimenti verrà lanciata un'eccezione `FileNotFoundException`.

Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il `FileDescriptor` relativo al file collegato allo stream.

Ecco ora un piccolo esempio di utilizzo di questi due stream per effettuare la copia di due file. Il nome dei due file (sorgente e destinazione) dovrà essere specificato sulla linea di comando.

```
import java.io.*;

public class CopyFile {
   static public void main (String args[]) throws IOException {
      if(args.length != 2){
```

```
         String  Msg;
         Msg = "Sintassi: CopyFile <sorgente> <destinazione>"
         throw(new IOException(Msg));
      }

      FileInputStream in = new FileInputStream(args[0]);
      FileOutputStream out = new FileOutputStream(args[1]);

      byte buffer[] = new byte[256];
      int n;
      while((n = in.read (buffer)) > -1)
         out.write(buffer, 0, n);

      out.close();
      in.close();
   }
}
```

Classi `Reader` e `Writer`

Dalla versione 1.1 del JDK, sono stati introdotti gli stream che gestiscono i caratteri (*character stream*). Tutti gli stream esaminati fino ad adesso gestiscono solo byte; i character stream sono come i byte stream, ma gestiscono caratteri Unicode a 16 bit, invece che byte (8 bit). Le classi base della gerarchia di questi stream sono `Reader` e `Writer`; tali classi supportano le stesse operazioni che erano presenti in `InputStream` e `OutputStream`, tranne che per il fatto che, laddove i byte stream operano su byte e su array di byte, i character stream operano su caratteri, array di caratteri, o stringhe.

Il vantaggio degli stream di caratteri è che rendono i programmi indipendenti dalla particolare codifica dei caratteri del sistema su cui vengono eseguite le applicazioni (a tal proposito si veda anche il capitolo sull'internazionalizzazione).

Java infatti per memorizzare le stringhe utilizza l'Unicode; l'Unicode è una codifica con la quale è possibile rappresentare la maggior parte dei caratteri delle varie lingue. I character stream quindi rendono trasparente la complessità di utilizzare le varie codifiche, mettendo a disposizione delle classi che automaticamente provvedono a eseguire la conversione fra gli stream di byte e gli stream di caratteri. La classe `InputStreamReader`, ad esempio, implementa un input stream di caratteri che legge i byte da un input stream di byte e li converte in caratteri. Allo stesso modo un `OutputStreamWriter` implementa un output stream di caratteri che converte i caratteri in byte e li scrive in un output stream di byte. Per creare un `InputStreamReader` basterà quindi eseguire la seguente operazione:

```
InputStreamReader in = new InputStreamReader(System.in);
```

Inoltre gli stream di caratteri sono più efficienti dei corrispettivi stream di byte, in

quanto, mentre questi ultimi eseguono spesso operazioni di lettura e scrittura un byte alla volta, i primi tendono a utilizzare di più la bufferizzazione.

A tal proposito esistono anche le classi `BufferedReader` e `BufferedWriter`, che corrispondo a `BufferedInputStream` e `BufferedOutputStream`; si può quindi scrivere

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

Le classi `PrintStream` e `PrintWriter`

Della classe `PrintStream` si è già parlato all'inizio del capitolo quando si sono introdotti gli stream predefiniti `in`, `out` ed `err`. Tale classe, nel JDK 1.1, è stata modificata in modo da utilizzare la codifica dei caratteri della piattaforma sottostante. Quindi in realtà ogni `PrintStream` incorpora un `OutputStreamWriter` e utilizza tale stream per gestire in modo adeguato i caratteri da stampare.

Invece di rendere *deprecated* l'intera classe `PrintStream`, sono stati resi deprecati i suoi costruttori. In questo modo tutti i programmi esistenti che, per stampare informazioni di debug o errori sullo schermo, utilizzano il metodo `System.out.println` o `System.err.println` potranno essere compilati senza ottenere warning. Si otterrà invece un warning se si costruisce esplicitamente un `PrintStream`.

In questi casi si dovrebbe invece costruire un `PrintWriter`, a cui si può passare un `OutputStream`, e che provvederà automaticamente a utilizzare un `OutputStreamWriter` intermedio per codificare in modo corretto i caratteri da stampare. I metodi per stampare sono quelli di `PrintStream` e cioè `print` e `println`, in grado di gestire i vari tipi primitivi di Java.

Altre classi e metodi deprecati

Quando si è trattato `DataInputStream` si è volutamente evitato il metodo `readLine`, per leggere una linea di testo dallo stream di input collegato, perché tale metodo è deprecated; questo è dovuto al fatto che non avviene la giusta conversione da byte a carattere. In tal caso si dovrebbe usare invece un `BufferedReader`, e il relativo metodo `readLine`.

Quindi, dato uno stream di input `in`, invece di creare un `DataInputStream`, se si vuole utilizzare il metodo `readLine`, si dovrà creare un `BufferedReader`:

```
BufferedReader d  = new BufferedReader(new InputStreamReader(in));
```

Comunque si potrà continuare a utilizzare la classe `DataInputStream` per tutte le altre operazioni di lettura.

Anche la classe `LineNumberInputStream`, utilizzata per tenere traccia delle linee all'interno di uno stream tramite il metodo `getLineNumber()`, è deprecata; al suo posto si dovrà utilizzare un `LineNumberReader`.

Capitolo 5

Networking

DI LORENZO BETTINI

Introduzione

Si sente spesso affermare che Java è "il linguaggio di programmazione per Internet". Effettivamente la maggior parte del grande successo e della diffusione di Java è dovuta a questo, vista soprattutto l'importanza sempre maggiore che Internet sta assumendo. Java è quindi particolarmente adatto per sviluppare applicazioni che devono fare uso della rete. Ciò non deve indurre a pensare che con Java si scrivono principalmente solo Applet, per animare e rendere più carine e interattive le pagine web. Con Java si possono sviluppare vere e proprie applicazioni che devono girare in rete interagendo con più computer (le cosiddette *applicazioni distribuite*).

Non si dimentichi che un altro fattore determinante per il suo successo è l'indipendenza dalla piattaforma, ottenuta grazie all'utilizzo del bytecode. Il linguaggio astrae da problemi di portabilità come il byte ordering, e quindi anche il programmatore non deve preoccuparsi dei classici problemi di interoperabilità cross-platform.

A questo punto il programmatore di una applicazione network based non deve preoccuparsi di scrivere ex novo particolari librerie o funzioni per le operazioni di base, ma può dedicarsi totalmente ai dettegli veri e propri dell'applicazione.

Inoltre ciò che rende Java un linguaggio adatto per il networking sono le classi definite nel pacchetto `java.net` che sarà analizzato in questo capitolo, in cui, oltre alla descrizione delle varie classi e dei rispettivi metodi, saranno forniti anche semplici esempi estendibili e funzionanti.

Socket

Le classi di networking incapsulano il paradigma *socket* presentato per la prima volta

nella Berkeley Software Distribution (BSD) della University of California at Berkeley.

Una socket è come una porta di comunicazione e non è molto diversa da una presa elettrica: tutto ciò che è in grado di comunicare tramite il protocollo standard TCP/IP può collegarsi ad una socket e comunicare tramite questa porta, allo stesso modo in cui un qualsiasi apparecchio che funziona a corrente può collegarsi a una presa elettrica e sfruttare la tensione messa a disposizione. Nella "rete" gestita dalle socket, invece dell'elettricità, viaggiano pacchetti TCP/IP. Tale protocollo e le socket forniscono quindi un'astrazione che permette di far comunicare dispositivi diversi che utilizzano lo stesso protocollo.

Quando si parla di networking, ci si imbatte spesso nel termine *client-server*. Si tratta in realtà di un paradigma: un'entità (spesso un programma) *client* per portare a termine un particolare compito richiede dei servizi ad un'altra entità (anche questa spesso è un programma): un *server* che ha a disposizione delle risorse da condividere. Una tale situazione si ritrova spesso nell'utilizzo quotidiano dei computer (anche senza saperlo): un programma che vuole stampare qualcosa (client) richiede alla stampante (server) l'utilizzo di tale risorsa.

Il server è una risorsa costantemente disponibile, mentre il client è libero di scollegarsi dopo che è stato servito. Tramite le socket inoltre un server è in grado di servire più client contemporaneamente.

Alcuni esempi di client-server molto noti sono:

Telnet : se sulla nostra macchina si ha disposizione il programma Telnet (programma client), è possibile operare su un computer remoto come si opera su un computer locale. Questo è possibile se sulla macchina remota è presente un programma server in grado di esaudire le richieste del client Telnet;

FTP : tramite un client FTP si possono copiare e cancellare files su un computer remoto, purché qui sia presente un server FTP;

Web : il browser è un client web, che richiede pagine web ai vari computer su cui è installato un web server, che esaudirà le richieste spedendo la pagina desiderata.

Come si è detto, tipicamente, sia il server che il client sono dei programmi che possono girare su macchine diverse collegate in rete. Il client deve conoscere l'indirizzo del server e il particolare protocollo di comunicazione utilizzato dal server. L'indirizzo in questione è un classico *indirizzo IP*.

Un client, quindi, per comunicare con un server usando il protocollo TCP/IP dovrà per prima cosa creare una socket con tale server, specificando l'indirizzo IP della macchi-

na su cui il server è in esecuzione e il numero di porta sulla quale il server è in ascolto. Il concetto di *porta* permette ad un singolo computer di servire più client contemporaneamente: su uno stesso computer possono essere in esecuzione server diversi, in ascolto su porte diverse. Se si vuole un'analogia si può pensare al fatto che più persone abitano nella medesima via, ma a numeri civici diversi. In questo caso i numeri civici rappresenterebbero le porte.

Un server "rimarrà in ascolto" su una determinata porta finché un client non creerà una socket con la macchina del server, specificando quella porta. Una volta che il collegamento con il server, tramite la socket è avvenuto, il client può iniziare a comunicare con il server, sfruttando la socket creata. A collegamento avvenuto si instaura un protocollo di livello superiore che dipende da quel particolare server: il client deve utilizzare quel protocollo di comunicazione, per richiedere servizi al server.

Il numero di porta è un intero compreso fra 1 e 65535. Il TCP/IP riserva le porte minori di 1024 a servizi standard. Ad esempio la porta 21 è riservata all'FTP, la 23 al Telnet, la 25 alla posta elettronica, la 80 all'HTTP (il protocollo delle pagine web), la 119 ai news server. Si deve tenere a mente che una porta in questo contesto non ha niente a che vedere con le porte di una macchina (porte seriali, parallele, ecc.), ma è un'astrazione utile per smistare informazioni a più server in esecuzione su una stessa macchina.

Si presentano adesso le classi messe a disposizione da Java nel pacchetto `java.net` per la gestione di comunicazioni in rete.

La classe `InetAddress`

Come si sa, un indirizzo Internet è costituito da 4 numeri (da 0 a 255) separati ciascuno da un punto. Spesso però, quando si deve accedere a un particolare host, invece di specificare dei numeri, si utilizza un nome, che corrisponde a tale indirizzo (p.e.: `www.myhost.it`). La traduzione dal nome all'indirizzo numerico vero e proprio è compito del servizio *Domain Name Service*, abbreviato con DNS.

Senza entrare nei dettagli di questo servizio, basti sapere che la classe `InetAddress` mette a disposizione diversi metodi per astrarre dal particolare tipo di indirizzo specificato (a numeri o a lettere), occupandosi essa stessa di effettuare le dovute traduzioni.

Inoltre c'è un ulteriore vantaggio: la scelta di utilizzare un indirizzo numerico a 32 bit non fu a suo tempo una scelta molto lungimirante; con l'immensa diffusione che Internet ha avuto e sta avendo, si è molto vicini ad esaurire tutti i possibili indirizzi che si possono ottenere con 32 bit (oltretutto diversi indirizzi sono riservati e quindi il numero di indirizzi possibili si riduce ulteriormente); si stanno pertanto introducendo degli indirizzi a 128 bit che, da questo punto di vista, non dovrebbero più dare tali preoccupazioni.

Le applicazioni che utilizzeranno indirizzi Internet tramite la classe `InetAddress` saranno portabili dal punto di vista degli indirizzi, in modo completamente trasparente.

Descrizione classe

```
public final class InetAddress extends Object implements Serializable
```

Costruttori

La classe non mette a disposizione nessun costruttore: l'unico modo per creare un oggetto `InetAddress` prevede l'utilizzo di metodi statici, descritti di seguito.

Metodi

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Restituisce un oggetto `InetAddress` rappresentante l'host specificato nel parametro `host`. L'host può essere specificato sia come nome, che come indirizzo numerico. Se si specifica `null` come parametro, ci si riferisce all'indirizzo di default della macchina locale.

```
public static InetAddress[] getAllByName(String host) throws UnknownHostException
```

Tale metodo è simile al precedente, ma restituisce un array di oggetti `InetAddress`: spesso alcuni siti web molto trafficati registrano lo stesso nome con indirizzi IP diversi. Con questo metodo si otterranno tanti `InetAddress` quanti sono questi indirizzi registrati.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Viene restituito un `InetAddress` corrispondente alla macchina locale. Se tale macchina non è registrata, oppure è protetta da un firewall, l'indirizzo è quello di loopback: `127.0.0.1`.

Tutti questi metodi possono sollevare l'eccezione `UnknownHostException` se l'indirizzo specificato non può essere risolto (tramite il DNS).

```
public String getHostName()
```

Restituisce il nome dell'host che corrisponde all'indirizzo IP dell'`InetAddress`. Se il nome non è ancora noto (ad esempio se l'oggetto è stato creato specificando un indirizzo

IP numerico), verrà cercato tramite il DNS; se tale ricerca fallisce, verrà restituito l'indirizzo IP numerico (sempre sotto forma di stringa).

```
public String getHostAddress()
```

Simile al precedente: restituisce però l'indirizzo IP numerico, sotto forma di stringa, corrispondente all'oggetto `InetAddress`.

```
public byte[] getAddress()
```

L'indirizzo IP numerico restituito sarà sotto forma di array byte. L'ordinamento dei byte è *high byte first* (che è proprio l'ordinamento tipico della rete).

Un'Applet potrà costruire un oggetto `InetAddress` solo per l'host dove si trova il web server dal quale l'Applet è stata scaricata, altrimenti verrà generata un'eccezione: `SecurityException`.

Un esempio

Con tale classe a disposizione è molto semplice scrivere un programma in grado di tradurre nomi di host nei corrispettivi indirizzi numerici e viceversa. Al programma che segue basterà passare una stringa contenente o un nome di host o un indirizzo IP numerico e si avranno in risposta le varie informazioni.

```
import java.net.*;
import java.io.*;

public class HostLookup {
   public static void main(String args[]) {
      // prima si stampano i dati relativi
      // alla macchina locale...
      try {
         InetAddress LocalAddress = InetAddress.getLocalHost();
         System.out.println("host locale : "
                           + LocalAddress.getHostName() + ", IP : "
                           + LocalAddress.getHostAddress());
      } catch(UnknownHostException e) {
         System.err.println("host locale sconosciuto!");
         e.printStackTrace();
      }

      // ...poi quelli dell'host specificato
```

```
      if(args.length != 1) {
         System.err.println("Uso: HostLookup host");
      } else {
         try {
            System.out.println("Ricerca di " + args[0] + "...");
            InetAddress RemoteMachine = InetAddress.getByName(args[0]);
            System.out.println("Host Remoto : "
                        + RemoteMachine.getHostName() + ", IP : "
                        + RemoteMachine.getHostAddress() );
         } catch(UnknownHostException ex) {
            System.out.println("Ricerca Fallita " + args[0]);
         }
      }
   }
}
```

URL

Tramite un URL (*Uniform Resource Locator*) è possibile riferirsi alle risorse di Internet in modo semplice e uniforme. Si ha così a disposizione una forma intelligente e pratica per identificare o indirizzare in modo univoco le informazioni su Internet. I browser utilizzano gli URL per recuperare le pagine web. Java mette a disposizione alcune classi per utilizzare gli URL; sarà così possibile, ad esempio, inglobare nelle proprie applicazioni funzioni tipiche dei web browser.

Esempi tipici di URL sono:

```
http://www.myweb.com:8080/webdir/webfile.html
ftp://ftp.myftpsite.edu/pub/programming/tips.tgz
```

Un URL consiste di 4 componenti:

1. il protocollo separato dal resto dai due punti (esempi tipici di protocolli sono `http`, `ftp`, `news`, `file`, ecc.);
2. il nome dell'host, o l'indirizzo IP dell'host, che è delimitato sulla sinistra da due barre (`//`), e sulla destra da una sola (`/`), oppure da due punti (`:`);
3. il numero di porta, separato dal nome dell'host sulla sinistra dai due punti, e sulla destra da una singola barra. Tale componente è opzionale, in quanto, come già detto, ogni protocollo ha una porta di default;
4. il percorso effettivo della risorsa che richiediamo. Il percorso viene specificato come si specifica un path sotto Unix. Se non viene specificato nessun file, la maggior parte dei server HTTP aggiunge automaticamente come file di default `index.html`.

Descrizione classe

```
public final class URL extends Object implements Serializable
```

Costruttori

La classe ha molti costruttori, poiché vengono considerati vari modi di specificare un URL.

```
public URL(String spec) throws MalformedURLException
```

L'URL viene specificato tramite una stringa, come ad esempio:

```
http://www.myweb.it:80/foo.html
```

```
public URL(URL context, String spec) throws MalformedURLException
```

L'URL viene creato combinando un URL già esistente e un URL specificato tramite una stringa. Se la stringa è effettivamente un URL assoluto, allora l'URL creato corrisponderà a tale stringa; altrimenti l'URL risultante sarà il percorso specificato in `spec`, relativo all'URL `context`. Ad esempio se `context` è `http://www.myserver.it/` e `spec` è `path/index.html`, l'URL risultante sarà `http://www.myserver.it/path/index.html`.

```
public URL(String protocol, String host, int port,String file) throws MalformedURLException
```

Con questo costruttore si ha la possibilità di specificare separatamente ogni singolo componente di un URL.

```
public URL(String protocol, String host, String file) throws MalformedURLException
```

Simile al precedente, ma viene usata la porta di default del protocollo specificato.

Un'eccezione `MalformedURLException` viene lanciata se l'URL non è specificato in modo corretto (per quanto riguarda i vari componenti).

Metodi

Di questa classe fanno parte diversi metodi che permettono di ricavare le varie parti di un URL.

```
public int getPort()
public String getProtocol()
public String getHost()
public String getFile()
```

Il significato di questi metodi dovrebbe essere chiaro: restituiscono un singolo componente dell'oggetto URL.

```
public String toExternalForm()
```

Restituisce una stringa che rappresenta l'URL

```
public URLConnection openConnection() throws IOException
```

Restituisce un oggetto `URLConnection` (sarà trattato di seguito), che rappresenta una connessione con l'host dell'URL, secondo il protocollo adeguato. Tramite questo oggetto, si può accedere ai contenuti dell'URL.

```
public final InputStream openStream() throws IOException
```

Apre una connessione con l'URL, e restituisce un input stream. Tale stream può essere utilizzato per leggere i contenuti dell'URL.

```
public final Object getContent() throws IOException
```

Questo metodo restituisce un oggetto di classe `Object` che racchiude i contenuti dell'URL. Il tipo reale dell'oggetto restituito dipende dai contenuti dell'URL: se si tratta di un'immagine, sarà un oggetto di tipo `Image`, se si tratta di un file di testo, sarà una `String`. Questo metodo compie diverse azioni, invisibili all'utente, come stabilire la connessione con il server, inviare una richiesta, processare la risposta, ecc.

Un esempio

Ovviamente per ogni protocollo ci dovrà essere un appropriato gestore. Il JDK fornisce di default un gestore del protocollo HTTP, e quindi l'accesso alle informazioni web è alquanto semplice.

Nel caso dell'HTTP, ad esempio una chiamata al metodo `openStream`, il gestore del protocollo HTTP, invierà una richiesta al web server specificato con l'URL, analizzerà le risposte del server, e restituirà un input stream dal quale è possibile leggere i contenuti del particolare file richiesto. Richiedere un file a un server web è molto semplice, ed è illustrato nell'esempio seguente, che mostra anche l'utilizzo di altri metodi della classe.

```
import java.net.*;
import java.io.*;

public class HTTP_URL_Reader {
   public static void main(String args[]) throws IOException {
      if(args.length < 1)
         throw new IOException("Sintassi : HTTP_URL_Reader URL");

      URL url = new URL(args[0]);

      System.out.println("Componenti dell'URL");
      System.out.println("URL    : " + url.toExternalForm());
      System.out.println("Protocollo: " + url.getProtocol());
      System.out.println("Host   : " + url.getHost());
      System.out.println("Porta     : " + url.getPort());
      System.out.println("File   : " + url.getFile());

      System.out.println("Contenuto dell'URL :");

      // lettura dei dati dell'URL
      InputStream iStream = url.openStream();
      DataInputStream diStream = new DataInputStream(iStream);

      String line ;
      while((line = diStream.readLine()) != null)
         System.out.println(line);

      diStream.close();
   }
}
```

È sufficiente creare un URL, passando al costruttore l'URL sotto forma di stringa, ottenere l'input stream chiamando l'apposito metodo, creare un `DataInputStream` basandosi su tale stream, e leggere una riga alla volta, stampandola sullo schermo. Il programma può essere eseguito così:

```
java HTTP_URL_Reader http://localhost/mydir/myfile.html
```

Se è installato un web server, si avranno stampate a schermo le varie componenti dell'URL specificato, nonché il contenuto del file richiesto.

La classe `URLConnection`

Questa classe rappresenta una connessione attiva, specifica di un dato protocollo, a un oggetto rappresentato da un URL. Tale classe è astratta, e quindi, per gestire uno specifico protocollo, si dovrebbe derivare da questa classe.

Descrizione classe

```
public class URLConnection extends Object
```

Costruttori

```
protected URLConnection(URL url)
```

Crea un oggetto di questa classe, dato un URL. Da notare che il costruttore è protetto, quindi può essere chiamato solo da una classe derivata. In effetti, come si è visto nella classe URL, un oggetto di questa classe viene ottenuto tramite la chiamata del metodo `openConnection` della classe `URL`.

Metodi

```
public URL getURL()
```

Restituisce semplicemente l'URL su cui è stato costruito l'oggetto `URLConnection`.

```
public abstract void connect() throws IOException
```

Permette di connettersi all'URL, specificato nel costruttore. La connessione quindi non avviene con la creazione dell'oggetto, ma avviene quando viene richiamato questo metodo, oppure un metodo che necessita che la connessione sia attiva (a quel punto la richiesta della connessione viene stabilita implicitamente).

```
public Object getContent() throws IOException
```

Restituisce il contenuto dell'URL. Viene restituito un oggetto di classe `Object`, poiché il tipo dell'oggetto dipende dal particolare URL.

```
public InputStream getInputStream() throws IOException
```

Restituisce un input stream con il quale si può leggere dall'URL.

```
public OutputStream getOutputStream() throws IOException
```

In questo caso si tratta di uno stream di output, con il quale è possibile inviare dati a un URL; ciò può risultare utile se si deve compiere un'operazione di post HTTP.

Questa classe contiene inoltre molti metodi che permettono di avere informazioni dettagliate sull'URL, quali il tipo di contenuto, la sua lunghezza, la data dell'ultima modifica, ecc. Per una rassegna completa si rimanda ovviamente alla documentazione on-line ufficiale.

Esistono poi alcuni metodi statici, da utilizzare per implementare gestori di protocollo personalizzati. Il trattamento di tale argomento va però oltre lo scopo di questo manuale.

I messaggi HTTP `GET` e `POST`

I web server permettono di ottenere informazioni come risultato di una query (interrogazione). Invece di richiedere un normale documento, si specifica nell'URL il nome di un programma (che segue l'interfaccia CGI), passandogli alcuni parametri che rappresentano la query vera e propria.

Molti sostengono che l'arrivo di Java abbia decretato la morte della programmazione CGI. In effetti tramite Java si ha più flessibilità, e i programmi vengono eseguiti dal lato client. Con la programmazione CGI invece il programma viene eseguito sul server, limitando così l'interazione con l'utente. Del resto il CGI è ancora molto usato, anche perché il web è pieno di programmi CGI già scritti e collaudati. Il presente paragrafo non vuole essere una spiegazione dettagliata della programmazione CGI (di cui non sarà data nessuna descrizione approfondita), ma vuol illustrare come dialogare con programmi CGI tramite Java.

Una query CGI è costituita quindi da un normale URL, con in coda alcuni parametri. La parte dell'URL che specifica i parametri inizia con un punto interrogativo (`?`). Ogni parametro è separato da una "e commerciale" (`&`), e i valori che si assegnano ai parametri sono specificati in questo modo: *`nome = valore`* (il valore è facoltativo). Un esempio di query è il seguente:

```
http://localhost/cgi-bin/mycgi.exe?nome=lorenzo&cognome=bettini&eta=29
```

In questo modo si richiama il programma CGI `mycgi.exe` e ad esso si passano i valori `lorenzo`, `bettini`, `29`, da assegnare rispettivamente ai parametri `nome`, `cognome`, `eta`.

Con la classe URL, presente nel pacchetto java.net, eseguire una tale query è semplicissimo: basta passare tale stringa al costruttore della classe.

Una query spedisce quindi dei dati al web server, inserendoli direttamente nell'URL. In questo modo però si può andare incontro a problemi dovuti alla limitatezza della lunghezza delle query: non si possono spedire grandi quantità di dati in questo modo.

Per far questo si deve utilizzare un altro messaggio HTTP: il messaggio POST. Infatti mentre un messaggio GET spedisce solo un header (intestazione) nel proprio messaggio, un messaggio POST, oltre che di un header, è dotato anche di un contenuto (content). Vale a dire che un messaggio POST è molto simile, strutturalmente, ad una risposta del server web, quando si richiede un documento (si veda a tal proposito l'esempio per ottenere una pagina web tramite le socket, nella sezione specifica). Infatti in un messaggio POST si deve includere il campo Content-length:.

Nel caso in cui si voglia inviare un messaggio POST si deve prima di tutto creare un oggetto URL, specificando un URL valido, creare un URLConnection con l'apposito metodo di URL, e abilitare la possibilità di utilizzare tale oggetto sia per l'output che per l'input. Inoltre è bene disabilitare la cache, in modo da essere sicuri che la risposta arrivi realmente dal server e non dalla cache.

```
URL destURL = new URL("http://localhost/cgi-bin/test-cgi");

URLConnection urlConn = destURL.openConnection();

urlConn.setDoOutput(true);
urlConn.setDoInput(true);
urlConn.setUseCaches(false);
```

Si deve poi riempire l'header del messaggio con alcune informazioni vitali, come il tipo del contenuto del messaggio e la lunghezza del messaggio, supponendo che il messaggio venga memorizzato in una stringa. Si ricordi che il contenuto deve essere sempre terminato da un \r\n.

```
String request = ...contenuto... + "\r\n";

urlConn.setRequestProperty("Content-type", "application/octet-stream");
urlConn.setRequestProperty("Content-length", "" + request.length());
```

A questo punto si può spedire il messaggio utilizzando lo stream dell'oggetto URLConnection (magari tramite un DataOutputStream). Dopo aver fatto questo ci si può mettere in attesa della risposta del server, sempre tramite lo stream (stavolta di input) di URLConnection.

```
DataOutputStream outStream
```

```
= new DataOutputStream(urlConn.getOutputStream());

outStream.writeBytes(request);
outStream.close();

DataInputStream inStream
= new DataInputStream(urlConn.getInputStream());

// lettura risposta dal server...
```

La classe `Socket`

Per creare una socket con un server in esecuzione su un certo host è sufficiente creare un oggetto di classe `Socket`, specificando nel costruttore l'indirizzo internet dell'host, e il numero di porta. Dopo che l'oggetto `Socket` è stato costruito è possibile ottenere (tramite appositi metodi) due stream (uno di input e uno di output). Tramite questi stream è possibile comunicare con l'host, e ricevere messaggi da esso. Qualsiasi metodo che prenda in ingresso un `InputStream` (o un `OutputStream`) può comunicare con l'host in rete.

Quindi, una volta creata la socket, è possibile comunicare in rete tramite l'usuale utilizzo degli stream.

Descrizione classe

```
public class Socket extends Object
```

Costruttori

```
public Socket(String host, int port) throws UnknownHostException, IOException

public Socket(InetAddress address, int port) throws IOException
```

Viene creato un oggetto `Socket` connettendosi con l'host specificato (sotto forma di stringa o di `InetAddress`) alla porta specificata. Se sull'host e sulla porta specificata non c'è un server in ascolto, verrà generata un'`IOException` (verrà specificato il messaggio `connection refused`).

Metodi

```
public InetAddress getInetAddress()
```

Restituisce un oggetto `InetAddress` corrispondente all'indirizzo dell'host con il quale la socket è connessa.

```
public InetAddress getLocalAddress()
```

Restituisce un oggetto `InetAddress` corrispondente all'indirizzo locale al quale la socket è collegata.

```
public int getPort()
```

Restituisce il numero di porta dell'host remoto con il quale la socket è collegata.

```
public int getLocalPort()
```

Restituisce il numero di porta locale con la quale la socket è collegata. Quando si crea una socket, come si è già detto, ci si collega con un server su una certa macchina, che è in ascolto su una certa porta. Anche sulla macchina locale, sulla quale viene creata la socket, si userà per tale socket una determinata porta, assegnata dal sistema operativo, scegliendo il primo numero di porta non occupato. Si deve ricordare infatti che ogni connessione TCP consiste sempre di un indirizzo locale e di uno remoto, e di un numero di porta locale e un numero di porta remoto. Questo metodo può essere utile quando un programma, già collegato con un server remoto, crei esso stesso un server. Per tale nuovo server può non essere specificato un numero di porta (a questo punto si prende il numero di porta assegnato dal sistema operativo). Con questo metodo si riesce a ottenere tale numero di porta, che potrà ad esempio essere comunicato ad altri programmi su altri host.

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Tramite questi metodi si ottengono gli stream, per mezzo dei quali è possibile comunicare attraverso la connessione TCP instaurata con la creazione della socket. Tale comunicazione sarà quindi basata sull'utilizzo degli stream, impiegati di continuo nella programmazione in Java. Come si può notare vengono restituite `InputStream` e `OutputStream`, che sono classi astratte. In realtà vengono restituiti dei `SocketInputStream` e `SocketOutputStream`, ma tali classi non sono pubbliche. Quando si comunica attraverso connessioni TCP, i dati vengono suddivisi in pacchetti (pacchetti IP appunto), quindi è consigliabile non utilizzare tali stream direttamente, ma sarebbe meglio costruire stream "bufferizzati" evitando così di avere pacchetti contenenti poche informazioni (infatti quando si inizia a scrivere i primi byte su tali stream, verranno spediti dei pacchetti con pochi byte, o forse anche un solo byte!).

```
public synchronized void close() throws IOException
```

Con questo metodo viene chiusa la socket (e quindi la connessione), e tutte le risorse che erano in uso verranno rilasciate. Dati contenuti nel buffer verranno comunque spediti, prima della chiusura del socket. La chiusura di uno dei due stream associati alla socket comporterà automaticamente la chiusura della socket stessa.

Può essere lanciata un'`IOException`, a significare che ci sono stati dei problemi sulla connessione. Ad esempio quando uno dei due programmi che utilizza la socket chiude la connessione, l'altro programma potrà ricevere una tale eccezione.

```
public synchronized void setSoTimeout(int timeout) throws SocketException
```

Dopo la chiamata di tale metodo, una lettura dall'`InputStream` della socket bloccherà il processo solo per una quantità di tempo pari a timeout (specificato in millisecondi). Se tale lasso di tempo scade, il processo riceverà un'`InterruptedIOException`. La socket rimane comunque valida e riutilizzabile. Se come `timeout` viene specificato `0`, l'attesa sarà illimitata (infinita), che è anche la situazione di default.

```
public synchronized int getSoTimeout() throws SocketException
```

Con questo metodo si può ottenere il timeout settato con il precedente metodo. Se viene restituito `0`, vuol dire che non è stato settato nessun timeout.

Quindi connettersi e comunicare con un server è molto semplice: basta creare una socket specificando host e porta (queste informazioni devono essere conosciute), ottenere e memorizzare gli stream della socket richiamando gli appositi metodi della socket, e utilizzarli per comunicare (sia per spedire informazioni, che per ricevere informazioni), magari dopo aver "bufferizzato" tali stream.

Utilizzo delle socket (client-server)

Si prenderà adesso in esame un semplice programma client: si tratta di un client HTTP che, dato un URL, richiede un file al server HTTP di quell'host. Si tratta di una variazione di `HTTP_URL_Reader` visto precedentemente durante la spiegazione della classe `URL`.

```
import java.net.*;
import java.io.*;

public class HTTPClient {
   public HTTPClient(String textURL) throws IOException {
```

```
      Socket socket = null;
      dissectURL(textURL);
      socket = connect();
      try {
         getPage();
      } finally {
         socket.close();
      }
   }

   protected String host, file;
   protected int port;

   protected void dissectURL(String textURL) throws MalformedURLException {
      URL url = new URL(textURL);
      host = url.getHost();
      port = url.getPort();
      if(port == -1)
         port = 80;
      file = url.getFile();
   }

   protected DataInputStream in;
   protected DataOutputStream out;

   protected Socket connect() throws IOException {
      System.err.println("Connessione a " + host + ":" + port + "...");
      Socket socket = new Socket(host, port);
      System.err.println("Connessione avvenuta.");

      BufferedOutputStream buffOut
        = new BufferedOutputStream(socket.getOutputStream());
      out = new DataOutputStream(buffOut);
      in = new DataInputStream(socket.getInputStream());

      return socket;
   }

   protected void getPage() throws IOException {
      System.err.println("Richiesta del file " + file + " inviata...");
      out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
      out.flush();

      System.err.println("Ricezione dati...");

      String input ;
      while((input = in.readLine()) != null)
         System.out.println(input);
   }
```

```
   public static void main(String args[]) throws IOException {
      if(args.length < 1)
         throw new IOException("Sintassi : HTTPClient URL");

      try {
         new HTTPClient(args[0]);
      } catch(IOException ex) {
         ex.printStackTrace();
      }

      System.out.println("exit");
   }
}
```

In effetti è stata ancora utilizzata questa classe per gestire l'URL passato sulla linea di comando, ma poi si effettua una connessione con il server creando esplicitamente una socket.

Nel metodo `connect` si effettua la connessione vera e propria aprendo una socket con l'host e sulla porta specificati:

```
Socket socket = new Socket(host, port);
```

Effettuata la connessione si possono ottenere gli stream associati con i metodi `getOutputStream` e `getInputStream` della classe `Socket`. Si crea poi un `DataOutputStream` e un `DataInputStream` su tali stream ottenuti (effettivamente, per ottimizzare la comunicazione in rete, prima viene creato uno stream "bufferizzato" sullo stream di output, ma questi dettagli, al momento, possono non essere approfonditi).

A questo punto si deve richiedere il file al server web e quindi si spedisce tale richiesta tramite lo stream di output:

```
out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
```

Ora non resta che mettersi in attesa, sullo stream di input, dell'invio dei dati dal server:

```
while((input = in.readLine()) != null)
```

Il contenuto del file viene stampato sullo schermo una riga alla volta.

Questo semplice programma illustra un esempio di client che invia al server una richiesta, e riceve dal server i dati richiesti. Questo è quanto avviene quando, tramite il proprio browser, si visita una pagina web: anche se in modo senz'altro più complesso il client apre una connessione con il server, comunica al server quello che desidera tramite un protocollo di comunicazione (nell'esempio, l'HTTP), e attende la risposta del server

(comunicata sempre tramite lo stesso protocollo). Il comando GET infatti fa parte del protocollo HTTP.

Per testare il programma non è necessaria una connessione a Internet, basta avere un web server installato e attivo e lanciare il programma in questo modo:

```
java HTTPClient http://localhost/index.html
```

E sullo schermo verrà stampato il contenuto dell'intero file index.html (se il file viene trovato, ovviamente, altrimenti si otterrà il tipico errore di file non trovato a cui ormai la navigazione web ci ha abituati).

Si vedrà adesso un esempio di programma *server*. Un server rimane in attesa di connessioni su una certa porta e, ogni volta che un client si connette a tale porta, il server ottiene una socket, tramite la quale può comunicare con il client. Il meccanismo messo a disposizione da Java per queste operazioni è la classe ServerSocket, tramite la quale il server può appunto accettare connessioni dai client attraverso la rete.

I passi tipici di un server saranno quindi:
1. creare un oggetto di classe ServerSocket specificando un numero di porta locale;
2. attendere (tramite il metodo accept() di suddetta classe) connessioni dai client;
3. usare la socket ottenuta ad ogni connessione, per comunicare con il client.
Infatti il metodo accept() della classe ServerSocket crea un oggetto Socket per ogni connessione. Il server potrà poi comunicare come fa un client: estraendo gli stream di input ed output dalla socket.

Tali passi possono essere riassunti nel seguente estratto di listato:

```
ServerSocket server = new ServerSocket(port);
Socket client = server.accept();
server.close();

InputStream i = client.getInputStream();
OutputStream o = client.getOutputStream();
```

Il server dell'esempio precedente chiude il ServerSocket appena ha ricevuto una richiesta di connessione, quindi tale server funziona una sola volta! Si ricorda che tale chiusura non chiude la connessione con il client appena creata: semplicemente il server non accetta ulteriori connessioni. Un server che "si rispetti", invece deve essere in grado

di accettare più connessioni e, inoltre, dovrebbe essere in grado di soddisfare più richieste contemporaneamente. Per risolvere questo problema si deve ricorrere al *multithreading*, per il quale Java offre diversi strumenti. Il programma sarà modificato nei due punti seguenti: il thread principale rimarrà in ascolto di richieste di connessioni; appena arriva una richiesta di connessione viene creato un thread che si occuperà di tale connessione e il thread principale tornerà ad aspettare nuove connessioni.

In effetti è questo quello che avviene nei server di cui si è già parlato. Se si osserva un programma scritto in C che utilizza le socket, si potrà vedere che appena viene ricevuta una richiesta di connessione, il programma si duplica (esegue una `fork()`), e il programma figlio lancia un programma che si occuperà di gestire la connessione appena ricevuta. Nel caso in esame basterà creare un nuovo thread e passargli la socket della nuova connessione.

Segue il programma modificato per trattare più connessioni contemporaneamente:

```
import java.net.*;
import java.io.*;

public class SimpleServer extends Thread {
   protected Socket client ;

   public SimpleServer(Socket socket) {

System.out.println("Arrivato un nuovo client da " + socket.getInetAddress()) ;
      client = socket;
   }

   public void run() {
      try {
         InputStream i = client.getInputStream();
         OutputStream o = client.getOutputStream();
         PrintStream p = new PrintStream(o);
         p.println("BENVENUTI.");
         p.println("Questo è il SimpleServer :-)");
         p.println();
         p.println("digitare HELP per la lista di servizi disponibili");

         int x;
         ByteArrayOutputStream command =  new ByteArrayOutputStream();
         String HelpCommand = new String("HELP");
         String QuitCommand = new String("QUIT");
         while((x = i.read()) > -1) {
            o.write(x);
            if(x == 13) { /* newline */
               p.println() ;
               if(HelpCommand.equalsIgnoreCase(command.toString())) {
                  p.println("Il solo servizio disponibile è l'help,");
```

```
               p.println("e QUIT per uscire.");
               p.println("Altrimenti che SimpleServer sarebbe... ;-)");
            } else if(QuitCommand.equalsIgnoreCase(command.toString())) {
               p.println("Grazie per aver usato SimpleServer ;-)");
               p.println("Alla prossima. BYE");
               try {
                  Thread.sleep(1000);
               } finally {
                  break;
               }
            } else {
               p.println("Comando non disponibile |-( ");
               p.println("Digitare HELP per la lista dei servizi");
            }
            command.reset();
         } else if( x != 10 ) /* carriage return */
            command.write(x);
      }
   } catch(IOException e) {
         e.printStackTrace();
   } finally {

System.out.println("Connessione chiusa con " + client.getInetAddress());
      try {
         client.close();
      } catch(IOException e) {
         e.printStackTrace();
      }
   }
}

public static void main(String args[]) throws IOException {
   int port = 0;
   Socket client;

   if(args.length == 1)
      port = Integer.parseInt(args[0]) ;

   System.out.println("Server in partenza sulla porta " + port);
   ServerSocket server = new ServerSocket(port);

System.out.println("Server partito sulla porta " + server.getLocalPort() );

   while(true) {
      System.out.println("In attesa di connessioni...");
      client = server.accept();

System.out.println("Richiesta di connessione da " + client.getInetAddress());
      (new SimpleServer(client)).start();
```

```
        }
    }
}
```

Questo programma accetta da linea di comando un parametro che specifica la porta su cui mettersi in ascolto di richieste di connessioni. Se non viene passato alcun argomento si userà la porta scelta dal sistema operativo. Dopo la creazione dell'oggetto `ServerSocket` ci si mette in ascolto di connessioni e, appena se ne riceve una, si fa partire un `Thread` per gestire tale connessione. In effetti tale classe deriva dalla classe `Thread`, e quindi, quando si crea un oggetto di questa classe, si crea in effetti un nuovo thread di esecuzione. In pratica si può riassumere:

- nel main si entra in un ciclo infinito (il ciclo finirà quando viene sollevata un'eccezione, oppure quando interrompiamo il programma), in cui viene eseguito `accept()`;
- appena viene ricevuta una richiesta di connessione si crea un nuovo oggetto della classe (e quindi un nuovo thread), passando ad esso la socket relativa a tale connessione, e viene lanciato così un nuovo thread di esecuzione;
- si torna ad eseguire l'`accept()`;
- il codice che si occupa della comunicazione con il client è nel metodo `run` che viene chiamato automaticamente quando un thread viene mandato in esecuzione (cioè quando si richiama il metodo `start()`).

Tramite l'oggetto `Socket` restituito dal metodo `accept` si ottengono i due stream per comunicare con il client. Si attende poi che il client invii dei comandi: ogni volta che viene letto un carattere, questo viene rispedito al client, in modo che quest'ultimo possa vedere quello che sta inviando. Appena viene digitato un `newline` (cioè *`invio`* o *`enter`*) si controlla se il servizio richiesto (memorizzato via via in un buffer) è disponibile, e si risponde in modo opportuno. Si noti come tutte le comunicazioni fra il server e il client siano racchiuse in un blocco `try-finally`: se nel frattempo avviene un'eccezione, si è comunque sicuri che la connessione verrà chiusa. L'eccezione in questione è tipicamente una `IOException` dovuta alla disconnessione da parte del client.

La classe deriva dalla classe `Thread`. È da notare come — poiché il metodo `run` della classe `Thread`, che viene ridefinito dalla nostra classe, non lancia nessuna eccezione — dobbiamo intercettare tutte le eccezioni all'interno del metodo: in questo caso l'eccezione in questione è `IOException` che può essere lanciata anche quando si cerca di chiudere la comunicazione. A proposito di client: in questo esempio, dov'è il client? Come nell'altro esempio avevamo usato un server già esistente (web server) per testare il nostro client, questa volta per testare il nostro server utilizzeremo un client classico: il *Telnet*.

Quindi se si è lanciato il server con la seguente riga di comando

```
java SimpleServer 9999
```

basterà utilizzare da un'altro terminale (ad esempio un'altra shell del DOS in Windows, o un altro xterm sotto Linux) il seguente comando:

```
telnet localhost 9999
```

Adesso è possibile inviare richieste al server semplicemente inserendo una stringa e premendo INVIO (provate ad esempio con "HELP").

User Datagram Protocol (UDP)

Finora si è sempre parlato del TCP (*Transfer Control Protocol*), un protocollo sviluppato sopra l'IP (*Internet Protocol*). Un altro protocollo basato sempre sull'IP, è l'UDP (*User Datagram Protocol*). In questo protocollo vengono spediti pacchetti di informazioni. Si tratta di un protocollo non basato sulla connessione (*connectionless*) e che non garantisce né l'arrivo né l'ordine dei pacchetti. Comunque, se i pacchetti arrivano, è garantito che siano integri e non corrotti. In un protocollo basato sulla connessione, come il TCP, si deve prima di tutto stabilire la connessione, dopo di che tale connessione può essere utilizzata sia per spedire che per ricevere. Quando la comunicazione è terminata, la connessione dovrà essere chiusa. Nell'UDP, invece, ogni messaggio sarà spedito come un pacchetto indipendente, che seguirà un percorso indipendente. Oltre a non garantire l'arrivo di tali pacchetti il protocollo non garantisce nemmeno che i pacchetti arrivino nell'ordine in cui sono stati spediti, e che non ci siano duplicati. Entrambi i protocolli utilizzano pacchetti, ma l'UDP, da questo punto di vista, è molto più vicino all'IP.

Ma perché utilizzare un protocollo così poco "affidabile"? Si tenga presente, che rispetto al TCP, l'UDP ha molto poco overhead (dovendo fare molti meno controlli), quindi può essere utilizzato quando la latenza è di fondamentale importanza. La perdita di pacchetti UDP è dovuta sostanzialmente alla congestione della rete. Utilizzando Internet questo è molto comune, ma se si utilizza una rete locale, questo non dovrebbe succedere.

Si può, comunque, aggiungere manualmente un po' di controllo sulla spedizione dei pacchetti. Si può supporre che, se non si riceve una risposta entro un certo tempo, il pacchetto sia andato perso, e quindi può essere rispedito. Va notato che, se il server ha ricevuto il pacchetto ma è la sua risposta che ha trovato traffico nella rete, il server riceverà nuovamente un pacchetto identico al precedente!

Si potrebbe allora pensare di implementare ulteriori controlli, ma questo porterebbe sempre più vicini al TCP. Nel caso in cui si voglia avere sicurezza sulla qualità di pacchetti, conviene passare direttamente al protocollo TCP appositamente pensato per questo scopo.

La classe `DatagramPacket`

Si devono creare oggetti di questa classe sia per spedire, sia per ricevere pacchetti. Un pacchetto sarà costituito dal messaggio vero e proprio e dall'indirizzo di destinazione. Per ricevere un pacchetto UDP si dovrà costruire un oggetto di questa classe e accettare un pacchetto UDP dalla rete. Non si possono filtrare tali pacchetti: si ricevono tutti i pacchetti UDP con il proprio indirizzo.

Descrizione classe

```
public final class DatagramPacket extends Object
```

Costruttori

```
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
```

Si costruisce un datagram packet specificando il contenuto del messaggio (i primi `ilength` bytes dell'array `ibuf`) e l'indirizzo IP del destinatario (sempre nella forma "host + numero porta").

Importante: poiché si tratta di protocolli differenti, un server UDP ed uno TCP possono essere in ascolto sulla stessa porta.

```
public DatagramPacket(byte ibuf[], int ilength)
```

In questo modo si costruisce un oggetto `DatagramPacket` da utilizzare per ricevere pacchetti UDP dalla rete. Il pacchetto ricevuto sarà memorizzato nell'array `ibuf` che dovrà essere in grado di contenere il pacchetto. `ilenght` specifica la dimensione massima di bytes che potranno essere ricevuti.

Metodi

Vi sono alcuni metodi che permettono di leggere le informazioni e il contenuto di un pacchetto UDP.

```
public InetAddress getAddress()
```

Se il pacchetto è stato ricevuto, tale metodo restituirà l'indirizzo dello host mittente; se l'oggetto `DatagramSocket` è invece stato creato per essere trasmesso, conterrà l'indirizzo IP del destinatario.

```
public int getPort()
```

Restituisce il numero del mittente o destinatario (vedi sopra), che può essere utilizzato per rispondere.

```
public byte[] getData()
```

Estrae dal pacchetto il contenuto del messaggio. L'array avrà la stesa grandezza specificata nel costruttore, e non l'effettiva dimensione del messaggio. Per ottenere tale dimensione si deve utilizzare il seguente metodo:

```
public int getLength()
```

La classe `DatagramSocket`

Questa classe permette di spedire e ricevere pacchetti UDP (sempre utilizzando le socket). Quando si spedisce un pacchetto UDP, come nel TCP, ci deve essere un `DatagramSocket` in ascolto sulla porta specificata.

Trattandosi di un protocollo connectionless, lo stesso oggetto `DatagramSocket` può essere utilizzato per spedire pacchetti a host differenti e ricevere pacchetti da host diversi.

Descrizione della classe

```
public class DatagramSocket extends Object
```

Costruttori

Si può specificare il numero di porta, oppure lasciare che sia il sistema operativo ad assegnarla. Tipicamente se si deve spedire un pacchetto (client) si utilizzerà la porta assegnata dal sistema operativo, e se si deve ricevere (server) si specificherà un numero di porta preciso. Ovviamente tale numero dovrà essere noto anche ai client. Ci sono quindi due costruttori:

```
public DatagramSocket() throws SocketException
```

```
public DatagramSocket(int port) throws SocketException
```

Si può inoltre specificare anche l'indirizzo al quale tale socket sarà legata:

```
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

Metodi

```
public void send(DatagramPacket p) throws IOException
```

Spedisce un pacchetto all'indirizzo di destinazione.

```
public synchronized void receive(DatagramPacket p) throws IOException
```

Riceve un singolo pacchetto UDP memorizzandolo in `p`. A questo punto si potranno ottenere tutte le informazioni su tale pacchetto, con i metodi della classe `DatagramPacket`.

```
public InetAddress getLocalAddress()
public int getLocalPort()
public synchronized void setSoTimeout(int timeout) throws SocketException
public synchronized int getSoTimeout() throws SocketException
public void close()
```

Questi metodi hanno lo stesso significato degli omonimi metodi della classe `Socket`; si rimanda quindi alla trattazione di tale classe.

Un esempio

Ecco un semplice esempio di utilizzo del protocollo UDP, tramite le due classi appena illustrate. Si tratta di due classi `UDPSender` e `UDPReceiver`, il cui nome dovrebbe essere esplicativo circa il loro funzionamento.

Ecco `UDPSender`:

```
import java.net.*;
import java.io.*;

public class UDPSender {
   static protected DatagramPacket buildPacket(String host, int port,
                                          String message) throws IOException {
      ByteArrayOutputStream boStream = new ByteArrayOutputStream();
      DataOutputStream doStream = new DataOutputStream(boStream);
      doStream.writeUTF(message);
      byte[] data = boStream.toByteArray();
```

```
      return new DatagramPacket(data, data.length,
                           netAddress.getByName(host), port);
   }

   public static void main(String args[]) throws IOException {
      if(args.length < 3)
         throw new IOException("Uso: UDPSender <host> <port> <messaggio> {messaggi}");

      DatagramSocket dsocket = new DatagramSocket();
      DatagramPacket dpacket ;

      for(int i = 2; i < args.length; i++) {
         dpacket = buildPacket(args[0], Integer.parseInt(args[1]), args[i]);
         dsocket.send(dpacket);
         System.out.println("Messaggio spedito");
      }
   }
}
```

e `UDPReceiver`:

```
import java.net.*;
import java.io.*;

public class UDPReceiver {
   static protected void showPacket(DatagramPacket p) throws IOException {
      System.out.println("Mittente  : " + p.getAddress());
      System.out.println("porta  : " + p.getPort()) ;
      System.out.println("Lunghezza messaggio : " + p.getLength());
      ByteArrayInputStream biStream
      = new ByteArrayInputStream(p.getData(), 0, p.getLength());
      DataInputStream diStream = new DataInputStream(biStream);
      String content = diStream.readUTF();
      System.out.println("Messaggio : " + content);
   }

   public static void main(String args[]) throws IOException {
      if(args.length != 1)
         throw new IOException("Uso: UDPReceiver <port>");

      byte buffer[] = new byte[65536];
      DatagramSocket dsocket
        = new DatagramSocket(Integer.parseInt(args[0]));
      DatagramPacket dpacket;

      while(true) {
         System.out.println("In attesa di messaggi...");
         dpacket = new DatagramPacket(buffer, buffer.length);
```

```
            dsocket.receive(dpacket);
            System.out.println("Ricevuto messaggio");
            showPacket(dpacket);
        }
    }
}
```

Si dovrà lanciare prima l'`UDPReceiver` specificando semplicemente il numero di porta su cui rimarrà in ascolto:

```
java UDPReceiver 9999
```

E su un altro terminale si potrà lanciare il sender specificando l'indirizzo e la porta, e poi una serie di stringhe, che verranno inviate al receiver:

```
java UDPSender localhost 9999 ciao a tutti
```

A questo punto il receiver mostrerà le informazioni riguardanti ogni messaggio ricevuto.

Su altri terminali si possono lanciare altri sender, sempre diretti allo stesso receiver, e si potrà notare che il receiver potrà ricevere messaggi da più sender, tramite lo stesso `DatagramSocket`. Non trattandosi di un protocollo con connessione, il socket rimarrà attivo anche quando i sender termineranno, cosa che non accade quando si crea una connessione diretta tramite una socket nel TCP.

Nuove estensioni e classi di utility presenti nella piattaforma Java 2

A partire dalla versione 2 del linguaggio sono state aggiunte al package `java.net` alcune classi di utilità che offrono un maggiore livello di astrazione o mettono a disposizione alcune *feature* ormai comuni nell'ambito del networking. Queste classi sono dedicate principalmente allo sviluppo di applicazioni che si appoggiano sul protocollo HTTP. Si vedranno qui di seguito, sinteticamente, alcune di queste classi con la descrizione dei principali metodi.

La classe `HttpURLConnection`

Estensione della classe `URLConnection`, questa classe mette a disposizione alcuni metodi specifici per il protocollo HTTP, metodi che permettono di tralasciare alcuni dettagli implementativi del protocollo stesso.

Il costruttore della classe ha la seguente firma

```
protected HttpURLConnection(URL myUrl)
```

Anche in questo caso, analogamente alla `URLConnection`, il costruttore è `protected`; per ottenere un oggetto di questa classe sarà sufficiente ricorrere allo stesso sistema con cui si ottiene un `URLConnection`, preoccupandosi di eseguire il `cast` opportuno.

Ad esempio si può scrivere

```
URL url = new URL(name);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
```

Metodi

```
public InputStream getErrorStream()
```

In caso di fallimento della connessione, permette di utilizzare lo stream restituito per ottenere le informazioni eventualmente inviate dal server sulle cause del fallimento.

```
public boolean getFollowRedirect()
```

Restituisce `true` se questa connessione è abilitata a seguire le redirezioni indicate dal server, `false` altrimenti (vedi oltre: `setFollowRedirect()`).

```
public Permission getPermission()
```

Restituisce un oggetto di tipo `Permission`, contenente i permessi necessari a eseguire questa connessione.

```
public String getRequestMethod()
```

Restituisce il metodo richiesto per questa connessione (`POST`, `GET`, ecc.) (vedi oltre: `setRequestMethod()`).

```
public int getResponseCode()
```

Restituisce il codice di stato della richiesta, inviato dal server.

```
public String getResponseMessage()
```

Restituisce il messaggio di risposta del server, collegato al codice.

```
public static void setFollowRedirect(boolean set)
```

Permette di configurare il comportamento di questa connessione a fronte di una richiesta di redirezione da parte del server.

```
public void setRequestMethod(String method) throws ProtocolException
```

Utilizzato per settare il metodo voluto per questa connessione. Il parametro è tipicamente una stringa indicante una delle operazioni previste dal protocollo HTTP, ad esempio `GET`, `POST`, ecc.; il metodo di default è `GET`.

La classe `JarURLConnection`

Questa classe astrae la connessione (HTTP) verso file archivio `.jar` remoti: il suo utilizzo si dimostra utile ad esempio nelle Applet, per accedere a file di immagini già presenti nella cache del browser.

Il meccanismo per ottenere un oggetto di tipo `JarURLConnection` è analogo a quello per `HttpURLConnection`; da notare in questo caso che, nella creazione della URL, è necessario specificare nel protocollo che si richiede una connessione ad un file `.jar`.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar");
```

Per ottenere tutto il file.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar/adirectory/afilename");
```

Per ottenere un file contenuto all'interno dell'archivio.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar/adirectory");
```

Per ottenere una directory contenuta nell'archivio.

Anche in questo caso per il costruttore

```
protected JarURLConnection(URL url)
```

vale quanto illustrato per la classe `HttpURLConnection`.

Metodi

```
public String getEntryName()
```

Restituisce la entry name se la connessione punta a un file contenuto in un archivio, `null` altrimenti.

```
public JarEntry getJarEntry()
```

Con questo metodo è possibile ottenere la `JarEntry` riferita all'oggetto della connessione; attraverso la `JarEntry` è possibile ottenere informazioni sull'oggetto della connessione, quali la dimensione in bytes, il metodo di compressione, ecc.

```
public JarFile getJarFile()
```

Restituisce il file `.jar` a cui fa riferimento questa connessione. Da notare che il file in questione non è modificabile.

```
public Manifest getManifest()
```

Restituisce, se esiste, il file `Manifest` contenuto nell'archivio.

Remote Method Invocation

di Giovanni Puliti

Introduzione

L'obiettivo principale della Remote Method Invocation (RMI) è quello di permettere a una applicazione in esecuzione su una macchina locale di invocare i metodi di un oggetto in esecuzione su un altro computer remoto. Anche se da un punto di vista architetturale lo schema è simmetrico, in genere si definisce il programma chiamante *client*, mentre il remoto è detto *server*.

Questo genere di programmi appartengono alla mondo della programmazione distribuita, indicando con tale termine un modo di organizzare una applicazione in moduli differenti localizzati in spazi di indirizzamento diversi fra loro.

Molti sono i vantaggi derivanti dall'adozione di tale modello: un sensibile miglioramento delle prestazioni complessive, una maggiore semplicità nella gestione delle risorse distribuite, ed un sostanziale incremento delle potenzialità operative. Ad esempio si può pensare di suddividere un processo computazionalmente pesante in sottoroutine più piccole ed eseguire tali "pezzi di applicazioni" su macchine diverse ottenendo una drastica diminuzione del tempo complessivo di esecuzione.

Nel caso in cui invece l'efficienza non sia l'obiettivo principale, si può comunque trarre vantaggio da una organizzazione distribuita, potendo gestire meglio e più semplicemente le varie risorse localizzate nei differenti spazi di indirizzamento. Si pensi per esempio ad una strutturazione a tre livelli (3-Tier) per la gestione di database relazionali in Internet: dal punto di vista del client ci si deve preoccupare esclusivamente dell'interfacciamento con l'utente e dello scambio con il server remoto delle informazioni contenute nel database.

Un altro importante motivo spinge all'utilizzo del modello distribuito, quando si debbano realizzare applicazioni che si interfacciano con codice legacy: in tal caso si può pensare di inglobare gli applicativi esistenti (*legacy* appunto) in oggetti remoti e pilotarne in

tal modo le funzionalità da client Java. In realtà per questo genere di integrazioni si preferisce spesso utilizzare tecnologie come CORBA, dato che RMI richiede l'utilizzo esclusivo di Java come linguaggio di sviluppo, cosa che rende difficile l'integrazione con programmi scritti in altri linguaggi.

Modelli distribuiti

Uno dei requisiti fondamentali per implementare un sistema distribuito è disporre di un sistema di comunicazione fra macchine diverse, basato su standard e protocolli prestabiliti.

In Java la gestione dei socket è un compito relativamente semplice, tanto che si possono realizzare in maniera veloce sistemi di comunicazione con i quali scambiare informazioni in rete o controllare sistemi remoti. L'implementazione di una connessione via socket risolve però solo il problema di fondo (come instaurare la connessione) ma lascia in sospeso tutta la parte di definizione delle modalità di invocazione e dei vari protocolli per lo scambio delle informazioni.

Prima dell'introduzione di RMI erano già disponibili strumenti per l'esecuzione di codice remoto, basti pensare alla *Remote Procedure Call* (RPC): con questa tecnologia è possibile gestire procedure facenti parte di applicazioni remote rispetto al chiamante. Le RPC hanno visto il massimo del loro successo nei sistemi Unix e sono strettamente legate al concetto di processo, ma male si inseriscono nel contesto del paradigma a oggetti. È questo il motivo principale che ha fatto nascere l'esigenza di una tecnologia apposita per la gestione di oggetti distribuiti, come RMI.

In realtà il panorama della progettazione e gestione di oggetti distribuiti offre valide alternative, come ad esempio DCOM (estensione di COM proprietaria di Microsoft) o CORBA: dato che una trattazione approfondita delle possibili alternative esula dagli scopi di questo capitolo, si può brevemente dire che la scelta può ricadere su RMI nel caso in cui si voglia implementare, in maniera semplice e veloce, una struttura a oggetti distribuiti *full-Java* (sia il lato client che quello server devono essere realizzati obbligatoriamente utilizzando tale linguaggio). Purtroppo la semplicità di RMI mostra spesso le sue limitazioni rispetto a tecnologie concorrenti più sofisticate (e complicate) che offrono maggiori potenzialità e garanzie di scalabilità.

La serializzazione

Il meccanismo alla base utilizzato da RMI per la trasmissioni dei dati fra client e server è quello della serializzazione: è quindi forse utile soffermarsi su questo importante sistema di trasmissione prima di affrontare nello specifico la Remote Method Invocation.

Grazie all'estrema semplicità con cui permette il flusso di dati complessi all'interno di uno stream, la serializzazione spesso viene utilizzata anche indipendentemente da applicazioni RMI, e quindi quanto verrà qui detto resta di utilità generale.

L'obiettivo principale della serializzazione è permettere la trasformazione automatica di oggetti e strutture di oggetti, in sequenze di byte manipolabili con i vari stream del package `java.io`.

Ad esempio, grazie alla serializzazione è possibile inviare oggetti di complessità arbitraria tramite un socket (utilizzando gli stream associati al socket stesso), oppure salvarli su file al fine di mantenere la persistenza.

La scrittura su stream avviene mediante il metodo `writeObject()` appartenente alla classe `ObjectOutputStream`. Ad esempio, volendo salvare su file una istanza di una ipotetica classe `Record`, si potrebbe scrivere

```
Record record = new Record();
FileOutputStream fos = new FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);
```

dove si è salvato su file binario (`data.ser`) un oggetto di tipo `Record`. L'operazione, in questo caso, è stata fatta in due fasi: creazione di uno stream di serializzazione prima, e associazione di tale stream a un comune `FileOutputStream`.

In modo altrettanto semplice si può effettuare l'operazione opposta che permette la trasformazione da stream ad oggetto

```
FileInputStream fis = new FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
ois.close();
```

In questo caso si utilizza la classe `ObjectInputStream` ed il metodo `readObject()`, il quale restituisce un oggetto di tipo `Object`, rendendo necessaria una operazione di conversione (cast esplicito).

In entrambi i casi le operazioni di lettura e scrittura devono essere inserite in appositi blocchi `try-catch` al fine di prevenire possibili problemi di lettura scrittura o di conversione.

Per poter serializzare un oggetto, un gruppo di oggetti, una struttura di complessità arbitraria, si utilizza sempre la medesima procedura e non ci sono particolari differenze di cui tener conto, a patto che l'oggetto sia serializzabile. Un oggetto è serializzabile se implementa l'interfaccia `Serializable`.

Per questo, ad esempio, l'oggetto `Record` visto nell'esempio di cui sopra potrebbe essere così definito:

```
public class Record implements Serializable {
   private String firstName;
   private String lastName;
   private int phone;
```

```
   public Record (String firstName, String lastName, int phone) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.phone = phone;
   }

}
```

La regola della serializzazione è ricorsiva, per cui un oggetto, per essere serializzabile, deve contenere esclusivamente oggetti serializzabili.

La maggior parte delle classi contenute all'interno del JDK è serializzabile, fatta eccezione per alcuni casi particolari: non sono serializzabili tutte le classi che inglobano al loro interno strutture dati binarie dipendenti dalla piattaforma, come ad esempio molti degli oggetti dell'API JDBC. In questo caso infatti i vari oggetti contengono al loro interno puntatori a strutture dati o codice nativo utilizzato per la comunicazione con lo strato driver del database.

Per sapere se un dato oggetto sia serializzabile o meno si può utilizzare il tool Serial Version Inspector (comando `serialver`), messo a disposizione dal JDK, passandogli il nome completo della classe da analizzare.

Ad esempio, per verificare che la classe `java.lang.String` sia serializzabile si può scrivere da linea di comando la seguente istruzione

```
serialver java.lang.String
```

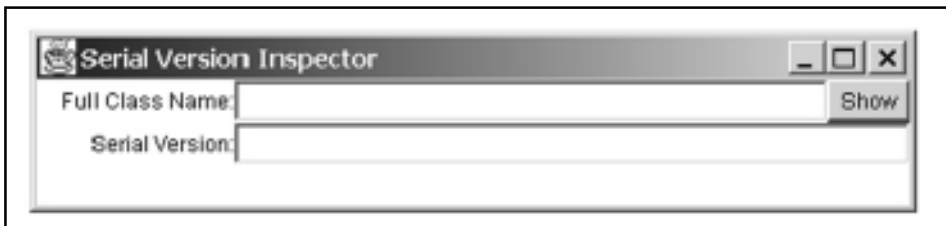
che restituisce il `serialVersionUID` dell'ogetto

```
java.lang.String: static final long serialVersionUID = -6849794470754667710L;
```

Invece tramite

```
serialver - show
```

si manda in esecuzione la versione con interfaccia grafica di tale strumento (vedi fig. 6.1).

Figura 6.1 – *Il tool Serial Version Inspector è disponibile anche in versione grafica*

La serializzazione e la trasmissione degli oggetti

Benché il suo utilizzo sia relativamente semplice, la serializzazione nasconde alcuni aspetti importanti relativamente alla trasmissione degli oggetti.

Per quanto visto finora si potrebbe immaginare che la serializzazione permetta la trasmissione di oggetti per mezzo di stream: in realtà questa concezione è quanto mai errata, dato che a spostarsi sono solamente le informazioni che caratterizzano un'istanza di un particolare oggetto.

Ad esempio durante la trasmissione in un socket non viene mai spostato fisicamente l'oggetto ma ne viene inviata solo la sua rappresentazione e successivamente viene ricreata una copia identica dall'altra parte del socket: al momento della creazione di questa copia il runtime creerà un oggetto nuovo, riempiendo i suoi dati con quelli ricevuti dal client.

Risulta ovvio quindi che, al fine di consentire questo spostamento virtuale, su entrambi i lati (sia server che client) debba essere presente il codice relativo all'oggetto: il runtime quindi deve poter disporre (o poter reperire in qualche modo) dei file `.class` necessari per istanziare l'oggetto.

Il `serialVersionUID` della classe serve proprio per identificare di quale tipo di oggetto siano i dati prelevati dallo stream. Si tenga presente che nella trasmissione delle informazioni relative all'oggetto sono inviati solamente quei dati realmente significativi della particolare istanza. Per questo non vengono inviati i metodi (che non cambiano mai), le costanti, le variabili con specificatore `static` che formalmente sono associate alla classe e non alla istanza, e quelle identificate con la parola chiave `transient`.

Con tale keyword si può qualificare una variabile non persistente, ovvero una variabile il cui valore non verrà inviato nello stream durante la serializzazione. Il valore assunto da una variabile di questo tipo dipende da come essa sia stata definita. Ad esempio, supponendo di scrivere

```
transient Integer  Int = new Integer(10);
```

al momento della deserializzazione alla variabile Int verrà impostato il valore 10. Se invece si fosse scritto

```
transient Integer  Int;
```

durante la fase di deserializzazione `Int` assumerebbe il proprio valore di default, che per tutte le variabili di tipo reference è null, mentre per i tipi primitivi corrisponde al valore base (0 per gli int, false per i boolean, 0.0 per i float e così via).

L'interfaccia `Externalizable`

Riconsiderando l'esempio visto in precedenza, la classe `Record` viene serializzata e deserializzato su uno stream. In questo caso il processo di trasformazione da oggetto a sequenza di byte è effettuato utilizzando le procedure standard di conversione del runtime.

Anche gli oggetti contenuti all'interno di `Record` sono trasformati in modo ricorsivo utilizzando le medesime tecniche.

Questa soluzione è molto potente e semplice, anche se in alcuni casi potrebbe essere utile ricorrere a procedure particolari di trasformazione. Ad esempio si potrebbe voler effettuare le operazioni di conversione su una delle variabili prima di effettuare il salvataggio su file di tutta la classe. In questo caso si può personalizzare il processo di serializzazione tramite l'interfaccia `Externalizable`.

Implementando tale interfaccia, ed in particolare ridefinendo il corpo dei due metodi `readExternal()` e `writeExternal()`, si potranno definire fin nei minimi dettagli tutti gli aspetti della conversione dell'oggetto.

Ad esempio si potrebbe definire

```
public class Record implements Externalizable {
   String Name;
   public MyObject(String n) {
      Name = n;
   }

   // salva i dati in modo personalizzato
   public void writeExternal(ObjectOutput out) {
      ...
   }

   // legge i dati in modo personalizzato
   public void readExternal(ObjectInput in) {
      ...
   }

}
```

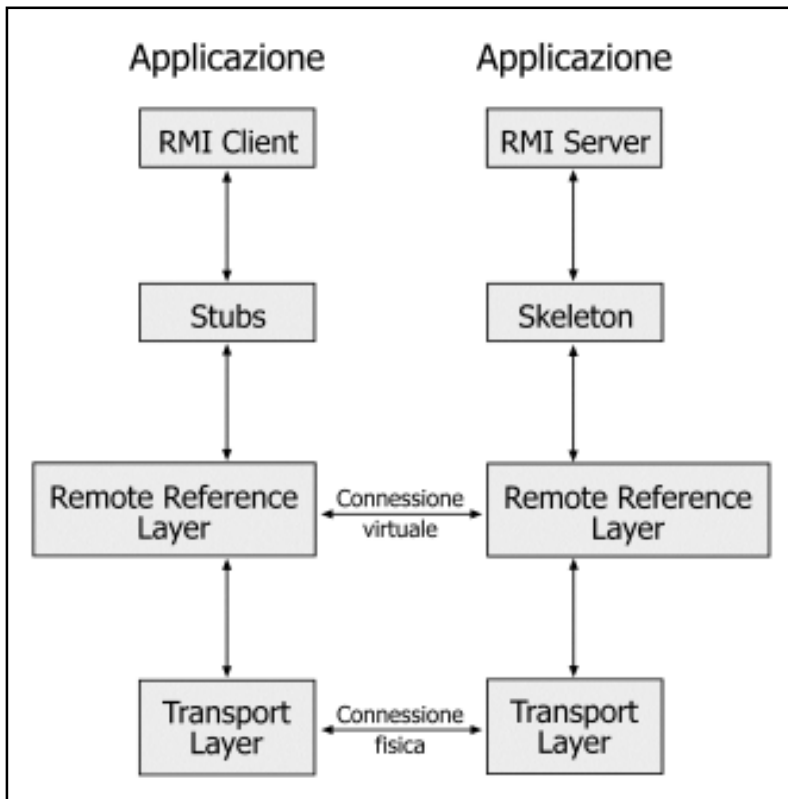
A questo punto l'oggetto `Record` potrà essere serializzato e deserializzato in maniera standard.

Architettura di RMI

Nella fig. 6.2 è riportata la struttura tipica di una applicazione RMI: è possibile notare come essa sia organizzata orizzontalmente in strati sovrapposti, e in due moduli verticali paralleli fra loro.

Questa suddivisione verticale vede da una parte il lato client, e dall'altra il server: il primo è quello che contiene l'applicazione che richiede il servizio di un oggetto remoto, che a sua volta diviene il servente del servizio RMI.

Lo strato più alto del grafico è costituito da entrambi i lati (client e server) da una applicazione eseguita sulla Java Virtual Machine in esecuzione su quel lato: nel caso del

Figura 6.2 – *Organizzazione a strati della architettura lato client e lato server di RMI*

client si tratta di una applicazione che effettua le chiamate ai metodi di oggetti remoti, i quali poi sono eseguiti dalla applicazione remota. Questa ha quindi un ciclo di vita indipendente dal client che di fatto ignora la sua presenza.

Subito sotto il livello "applicazione" troviamo i due elementi fondamentali dell'architettura RMI, ovvero lo stub e lo skeleton. Questi due oggetti forniscono una duplice rappresentazione dell'oggetto remoto: lo stub rappresenta una simulazione locale sul client dell'oggetto remoto che però, grazie allo skeleton, vive e viene eseguito sul lato server; i due elementi quindi non sono utilizzabili separatamente.

Da un punto di vista funzionale, il client, dopo aver ottenuto un reference dell'oggetto remoto, lo stub di tale oggetto, ne esegue i metodi messi a disposizione per l'invocazione remota, in modo del tutto analogo al caso in cui l'oggetto sia locale. Si può quindi scrivere

```
OggettoRemoto.nomeMetodo();
```

Da un punto di vista sintattico non vi è quindi nessuna differenza fra un oggetto locale e uno remoto.

In conclusione il client (intendendo sia il programma che il programmatore della applicazione lato client) non ha che in minima parte la percezione di utilizzare un oggetto remoto.

Passaggio di parametri in RMI

Uno dei grossi vantaggi nell'utilizzo di RMI consiste nella semplicità con cui si possono passare parametri durante l'invocazione dei metodi remoti e riceverne indietro i risultati: senza nessuna differenza rispetto al caso locale si può scrivere

```
Ris  = OggettoRemoto.nomeMetodo(param_1, ..., param_n);
```

Riconsiderando lo schema riportato nella figura 2, si ha che i vari parametri vengono serializzati dalla virtual machine del client, inviati sotto forma di stream al server, il quale li utilizzerà in forma deserializzata per utilizzarli all'interno del corpo del metodo invocato. Anche il percorso inverso, ovvero quello che restituisce un risultato al client, effettua una serializzazione e quindi una deserializzazione dei dati.

Il procedimento, che da un punto di vista teorico risulta essere piuttosto complesso, è semplicissimo da utilizzare. L'unica reale avvertenza di cui si deve tener conto è che i parametri passati e il risultato ricevuto siano oggetti serializzabili.

Si tenga presente che, durante la trasmissione bidirezionale dei dati, viene sempre effettuata una copia (clonazione) dei reference, per cui non si ha la possibilità di lavorare su aree di memoria fisse, esigenza questa del resto non necessaria dato che si opera in uno scenario distribuito.

Gli strati RRL e TL

Il lato server e client sono collegati col sottostante Remote Reference Layer (RRL) che a sua volta si appoggia al Transport Layer (TL).

Il primo dei due ha il compito di instaurare un collegamento logico fra i due lati, di codificare le richieste del client, inviarle al server, decodificare le richieste e inoltrarle allo skeleton.

Ovviamente nel caso in cui quest'ultimo fornisca dei risultati per il particolare tipo di servizio richiesto, il meccanismo di restituzione di tali valori avviene in maniera del tutto simile ma in senso opposto.

Al livello RRL viene instaurato un collegamento virtuale fra i due lati client e server, mentre fisicamente la connessione avviene al livello sottostante, quello definito Transport Layer. Tale collegamento è di tipo sequenziale ed è per questo che si richiede la serializzazione dei parametri da passare ai metodi.

Il collegamento virtuale del RRL si basa su un protocollo di comunicazione generico e indipendente dal particolare tipo di stub o skeleton utilizzati: questa genericità permette di mantenere la massima indipendenza dal livello stub/skeleton, tanto che è possibile sostituire il RRL con versioni successive più ottimizzate.

Il protocollo di conversione delle invocazioni dei metodi, l'impacchettamento dei riferimenti ai vari oggetti, e tutto quello che concerne la gestione a basso livello, sono operazioni a carico sia dello strato RRL, sia e soprattutto dal TL, in cui si perde la concezione di oggetto remoto e/o locale e i dati vengono semplicemente visti come sequenze di byte da inviare o leggere verso certi indirizzi di memoria.

Quando il TL riceve una richiesta di connessione da parte del client, localizza il server RMI relativo all'oggetto remoto richiesto: successivamente viene eseguita una connessione per mezzo di un socket appositamente creato per il servizio. Una volta che la connessione è stabilita, il TL passa la connessione al lato client del RRL e aggiunge un riferimento dell'oggetto remoto nella tabella opportuna. Solo dopo questa operazione il client risulta effettivamente connesso al server e lo stub è utilizzabile dal client.

Il TL è responsabile del controllo dello stato delle varie connessioni: se passa un periodo di tempo significativo senza che venga effettuato nessun riferimento alla connessione remota, si assume che tale collegamento non sia più necessario, e quindi viene disattivato. Mediamente il periodo di timeout scatta dopo 10 minuti.

L'ultimo livello che però non viene incluso nella struttura RMI, è quello che riguarda la gestione della connessine al livello di socket e protocolli TCP/IP. Questo aspetto segue le specifiche standard di networking di Java e non offre particolari interessanti in ottica RMI.

RMI in pratica

Si può ora procedere ad analizzare quali siano i passi necessari per realizzare una applicazione RMI. Tutte le classi e i metodi che si analizzeranno, e in generale tutte le API necessarie per lavorare con RMI, sono contenute nei package `java.rmi` e `java.rmi.server`.

Anche se dal punto di vista della programmazione a oggetti sarebbe più corretto parlare di classi, in questo caso si parlerà genericamente di oggetti remoti e locali intendendo sia il tipo che la variabile.

A tal proposito, in base alla definizione ufficiale, si definisce remoto un oggetto che implementi l'interfaccia `Remote` ed i cui metodi possano essere eseguiti da una applicazione client non residente sulla stessa macchina virtuale.

Un'interfaccia remota invece rende disponibile il set di metodi utilizzabili per l'invocazione a distanza; ovviamente non è necessario definire nell'interfaccia quei metodi a solo uso interno della classe. Si immagini quindi di definire `MyServer` un oggetto per il momento non remoto.

```
public class MyServer {
   public void String concat(String a, String b) {
      return a + b;
   }
}
```

Il metodo `concat()` in questo caso esegue una concatenazione fra i due argomenti passati in input restituendo in uscita la stringa risultante.

A parte il vincolo della serializzabilità dei parametri, non ci sono limiti alla complessità delle operazioni eseguibili all'interno di metodi remoti.

Dopo aver definito questa semplice classe per trasformarla nella versione remota si deve per prima cosa definire la sua interfaccia remota.

```
public interface MyServerInterface extends Remote {
   public String concat(String a, String b) throws RemoteException;
}
```

Come si può osservare da queste poche righe di codice, per definire un'interfaccia remota è necessario estendere la `java.rmi.Remote`: questa è una interfaccia vuota e serve solo per verificare durante l'esecuzione che le operazioni di invocazione remota siano plausibili.

È obbligatoria la gestione dell'eccezione `java.rmi.RemoteException`: infatti, a causa della distribuzione in rete, oltre alla gestione di eventuali problemi derivanti dalla normale esecuzione del codice (bug o incongruenze di vario tipo), si deve adesso proteggere tutta l'applicazione da anomalie derivanti dall'utilizzo di risorse remote. Ad esempio potrebbe venire a mancare improvvisamente la connessione fisica verso l'host dove è in esecuzione il server RMI.

Definita l'interfaccia remota si deve modificare leggermente la classe di partenza, in modo che implementi questa interfaccia:

```
public class MyServerImpl implements MyServerInterface
                             extends UnicastRemoteObject {

   public MyServerImpl() throws RemoteException {
      …
   }

   public String concat(String a, String b) throws RemoteException{
      return a + b;
   }
}
```

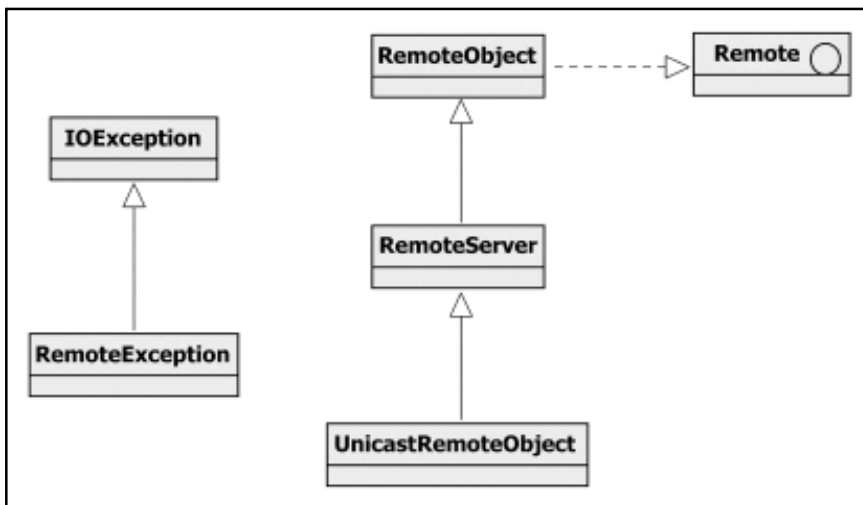
Il nome della classe è stato cambiato a indicare l'implementazione della interfaccia remota; come si può notare oltre a dichiarare di implementare l'interfaccia precedentemente definita, si deve anche estendere la classe `UnicastRemoteObject`.

Oltre a ciò, all'oggetto è stato aggiunto un costruttore di default il quale dichiara di propagare eccezioni `RemoteException`: tale passaggio non ha una motivazione apparente, ma è necessario per permettere al compilatore di creare correttamente tutte le parti che compongono il lato server (stub e skeleton).

La classe `UnicastRemoteObject` deriva dalle due classi, `RemoteServer` e `RemoteObject`: la prima è una superclasse comune per tutte le implementazioni di oggetti remoti (la parola `Unicast` ha importanti conseguenze come si avrà modo di vedere in seguito), mentre l'altra semplicemente ridefinisce `hashcode()` ed `equals()` in modo da permettere correttamente il confronto tra oggetti remoti.

L'utilizzazione della classe `RemoteServer` permette di utilizzare implementazioni di oggetti remoti diverse da `UnicastRemoteObject`, anche se per il momento quest'ultima è l'unica supportata.

L'organizzazione delle più importanti classi per RMI è raffigurata in fig. 6.3.

Figura 6.3 – *Struttura gerarchica delle classi ed interfacce più importanti in RMI*

Dopo questa trasformazione l'oggetto è visibile dall'esterno, ma ancora non utilizzabile secondo la logica RMI: si devono infatti creare i cosiddetti stub e skeleton.

Tali oggetti sono ottenibili in maniera molto semplice per mezzo del compilatore `rmic`, disponibile all'interno del JDK 1.1 e successivi: partendo dal bytecode ottenuto dopo la compilazione dell'oggetto remoto, questo tool produce lo stub e lo skeleton relativi. Ad esempio, riconsiderando il caso della classe `MyServerImpl`, con una operazione del tipo

```
rmic MyServerImpl
```

si ottengono i due file `MyServerImpl_stub.class` e `MyServerImpl_skel.class`.

A questo punto si hanno a disposizione tutti i componenti per utilizzare l'oggetto remoto `MyServerImpl`: resta quindi da rendere possibile il collegamento tra client e server per l'invocazione remota.

Si definisce server RMI l'applicazione che istanzia un oggetto remoto e lo registra tramite una bind all'interno dell'rmiregistry. Il server RMI non è quindi l'oggetto che implementa la business logic, ma solamente una applicazione di servizio necessaria per attivare il meccanismo di invocazione remota.

Sul lato server l'applicazione che gestisce lo skeleton deve notificare che possiede al suo interno un oggetto abilitato all'invocazione remota. Per far questo è necessario utilizzare il metodo statico `java.rmi.Naming.bind()` che associa all'istanza dell'oggetto remoto un nome logico con cui tale oggetto può essere identificato in rete.

Quindi, dopo aver creato una istanza dell'oggetto remoto,

```
MyServer server = new MyServer();
```

si provvede a effettuarne la registrazione utilizzando un nome simbolico

```
Naming.bind("MyServer", server);
```

Questa operazione, detta registrazione, può fallire e in tal caso viene generata una eccezione in funzione del tipo di errore. In particolare si otterrà una `AlreadyBoundException` nel caso in cui il nome logico sia già stato utilizzato per un'altra associazione, una `MalformedURLException` per errori nella sintassi dell'URL, mentre il runtime produrrà `RemoteException` per tutti gli altri tipi di errore legati alla gestione da remoto dell'oggetto.

Ogni associazione *nome logico – oggetto remoto* è memorizzata in un apposito registro detto RMIRegistry. In questo caso `rmiregistry` è anche il comando che lancia l'applicazione per gestire tale archivio, applicazione che deve essere lanciata sul lato server prima di ogni bind. Il client a questo punto è in grado di ottenere un reference all'oggetto con una ricerca presso l'host remoto utilizzando il nome logico con cui l'oggetto è stato registrato.

Ad esempio si potrebbe scrivere

```
MyServerInterface server;
String url = "//" + serverhost + "/MyServer";
```

```
server = (MyServerInterface) Naming.lookup(url);
```

e quindi utilizzare il reference per effettuare le invocazioni ai metodi remoti

```
System.out.println(server.concat("Hello ", "world!"));
```

Sul client per ottenere il reference si utilizza il metodo statico `Naming.lookup()`, che può essere considerato il corrispettivo alla operazione di bind sul server.

L'URL passato come parametro al `lookup()` identifica il nome della macchina che ospita l'oggetto remoto e il nome con cui l'oggetto è stato registrato.

Entrambe le operazioni di registrazione e di ricerca accettano come parametro un URL: il formato di tale stringa è la seguente:

```
rmi://host:port/name
```

dove *host* è il nome del server RMI, *port* la porta dove si mette in ascolto il registry, *name* il nome logico.

Sul server non è necessario specificare l'host, dato che per default assume l'indirizzo della macchina stessa sulla quale l'applicazione server RMI è mandata in esecuzione.

In entrambi i casi il numero della porta di default è la 1099, ma se si specifica altrimenti, allora tale informazione dovrà essere passata al `rmiregistry` con il seguente comando:

```
rmiregistry numero_porta
```

Download del codice da remoto

Ogni volta che un parametro viene passato ad un metodo remoto, o viceversa ogni volta che si preleva un oggetto come risultato di una computazione remota, si da vita ad un processo di serializzazione o deserializzazione dell'oggetto in questione.

In realtà, come si è potuto vedere, l'oggetto serializzato non viene spostato dal client al server, ma vengono inviate nella rete solamente le informazioni necessarie per ricreare una copia dell'oggetto dal client al server (e viceversa).

Questo significa che sia il client che il server devono poter disporre dello stesso bytecode relativo all'oggetto serializzato in modo da poterne ricreare l'istanza. La soluzione più semplice è copiare fisicamente i vari file .class sia sul server che sul client: in questo caso si potrà essere sicuri che le varie operazioni di serializzazione e deserializzazione potranno essere effettuate correttamente.

Lo svantaggio di questa organizzazione risiede nel dover, per ogni modifica delle varie classi, ridistribuire tutti i file. In alcuni casi questa soluzione è scomoda se non addirittura impraticabile.

RMI mette a disposizione un meccanismo molto potente che permette di scaricare dalla rete, tramite un server HTTP, i file necessari per il funzionamento del client.

Nel caso delle applet il classloader effettua una chiamata al motore HTTP del browser per scaricare tali file: da questo punto di vista non vi è differenza fra oggetti remoti che devono essere presenti in locale per la deserializzazione, e le normali classi Java necessarie per far funzionare l'applet.

Le classi sono localizzate automaticamente in Internet, facendo riferimento al codebase di provenienza (il quale tra l'altro è l'unico indirizzo verso il quale l'applet può connettersi).

Per quanto riguarda invece le applicazioni, la mancanza del browser complica le cose, dato che devono essere risolti due problemi: il primo è come effettuare il download vero e proprio delle classi, e il secondo come localizzare il server dal quale scaricare tale codice.

Per il primo aspetto non ci si deve preoccupare più di tanto, dato che esiste un oggetto apposito che effettua tale operazione: si tratta della classe `RMIClassLoader` parte integrante del package `rmi.server`, e che può essere vista come una evoluzione della `ClassLoader`.

Per specificare l'indirizzo dell'host remoto dove è in funzione tale server si può procedere in due modi: o lo si inserisce direttamente dentro il codice (hardcoded URL), oppure lo si passa al client come parametro di configurazione dall'esterno. Per rendere le cose ancora più semplici ed automatiche si può fare in modo che sia il server, utilizzando il protocollo RMI, a comunicare al client l'indirizzo dove prelevare tali classi (nel caso delle applicazioni, non essendoci particolari vincoli, tale indirizzo potrà essere differente da quello server RMI). Per fare questo è necessario mandare in esecuzione il server RMI specificando nella opzione `java.rmi.server.codebase` l'URL presso cui prelevare via HTTP le classi necessarie.

La sintassi di esecuzione del server è la seguente

```
java -Djava.rmi.server.codebase = http://nome_host:port/rmi_dir/ ServerRmi
```

dove *`ServerRmi`* indica il nome della applicazione server RMI, mentre *`nome_host`*`:`*`port`* specifica l'indirizzo Internet e la porta dove è in funzione il server HTTP.

Tale demone dovrà avere accesso alle classi remote che dovranno essere posizionate nella directory *`rmi_dir`*`/`.

Download del codice e sicurezza

Nell'ambito della programmazione di rete, nasce il problema di garantire un sufficiente livello di sicurezza tutte le volte che si esegue del codice scaricato da remoto: chi garantisce che tale codice non esegua operazioni pericolose?

Java effettua in tal senso un controllo molto scrupoloso, grazie alla classe `SecurityManager` che, nel caso di RMI si chiama `RMISecurityManager`. Dal punto di vista del client, se nessun `RMISecurityManager` è stato installato, allora potranno essere caricate classi solamente dal classpath locale.

RMI e la programmazione per pattern

L'estrema semplicità con cui è possibile realizzare applicazioni distribuite in Java permette sicuramente di semplificare il lavoro in maniera notevole, ma spesso si dimostra essere anche la sua principale limitazione.

Alcuni dei vincoli imposti dal modello base di RMI possono essere superati in maniera piuttosto brillante grazie a una opportuna progettazione della struttura dell'applicazione. In particolare la programmazione per pattern ([GOF] [java pattern] [mokabyte pattern]), permette di dar vita a interessanti soluzioni, e proprio per questo sarà preso in esame il pattern Factory Method.

RMI e il pattern Factory

Se si ripensa per un momento alla modalità di pubblicazione di un oggetto remoto da parte del server RMI, si potrà osservare come la funzione di creazione e registrazione sia un compito totalmente a carico del server. Per una precisa scelta progettuale quindi, visto che la registrazione dell'oggetto avviene una volta sola, l'istanza dell'oggetto remoto sarà l'unica disponibile e quindi condivisa fra tutti i client possibili. Pensata per la massima semplicità possibile, in molti casi però questa soluzione risulta essere troppo rigida e non sufficiente per supportare architetture distribuite complesse.

Si pensi ad esempio agli oggetti remoti come a veri e propri servizi messi a disposizione dal server e per questo per brevità verranno qui chiamati “oggetti remoti di servizio”.

Per rendere tali oggetti pubblici e invocabili dai vari client RMI, si potrebbe pensare di crearne un certo numero, e successivamente di registrarli con nomi differenti.

Ovviamente tale soluzione non risolve il problema della condivisione fra i vari client, dato che per tanti che possano essere gli oggetti remoti di servizio registrati, i client potrebbero essere in numero maggiore; tale organizzazione non risolve inoltre il problema della creazione on demand da parte del client.

Il modello di RMI prevede una soluzione a tale problema, ma risulta essere alquanto complessa (vedi oltre): utilizzando il pattern Factory si può ottenere lo stesso risultato in modo molto più semplice ed elegante.

Si supponga ad esempio di avere un'insieme base di oggetti strutturati come riportato in fig. 6.4. Si tratta di una semplice gerarchia di classi per la gestione dei messaggi di log all'interno di una applicazione: per semplicità verrà preso in considerazione il caso in cui sia presente il solo metodo `log(String` *`messaggio`*`)` per la produzione di tali messaggi.

Figura 6.4 – *Struttura gerarchica di base per la gestione di messaggi di log*

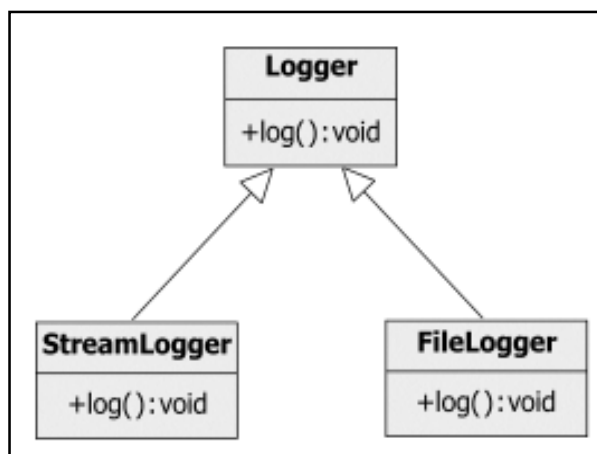
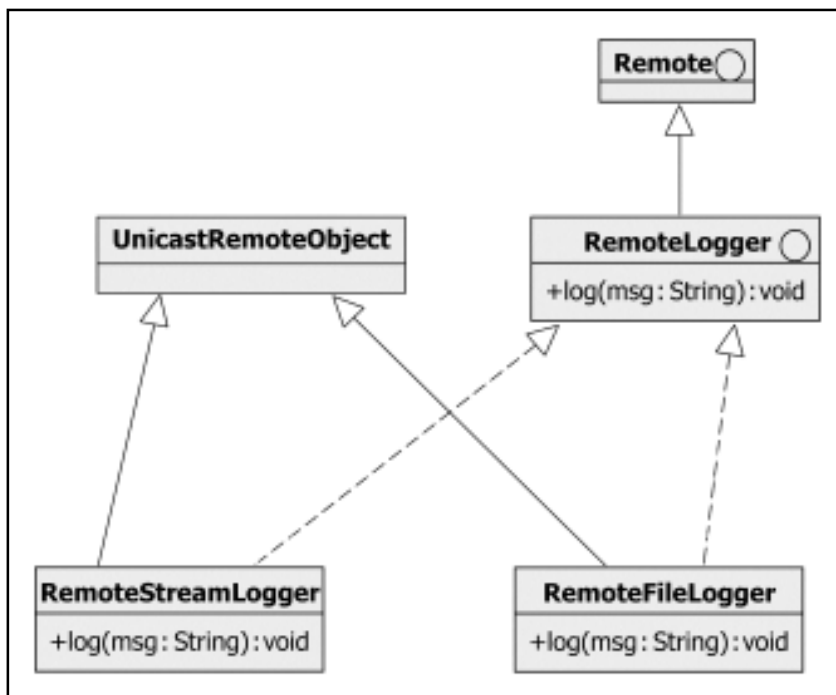



Figura 6.5 – *Versione remota delle classi per la gestione dei messaggi da client RMI*

La classe base (`Logger`) serve pertanto a gestire messaggi di tipo generico, mentre le due classi derivate potrebbero implementare tecniche particolari per la gestione di messaggi verso file (classe `FileLogger`) e verso uno stream generico (classe `StreamLogger`).

A tal punto si può immaginare che un ipotetico client possa aver bisogno di utilizzare i servizi di un oggetto di tipo `FileLogger` oppure `StreamLogger`, ad esempio per memorizzare alcune informazioni relative al suo funzionamento.

In uno scenario reale si potrebbe ipotizzare che ogni client debba o voglia poter produrre messaggi propri indipendentemente dagli altri, e che tali messaggi siano gestiti da un server centrale.

Utilizzando RMI allora si dovranno per prima cosa creare *n* oggetti remoti (in teoria uno per ogni client che desideri fare log da remoto) e successivamente registrarli.

Per rendere remoti gli oggetti visti in precedenza è necessario modificare leggermente la gerarchia di classi, cosa che può portare a una organizzazione delle classi come quella riportata in fig. 6.5.

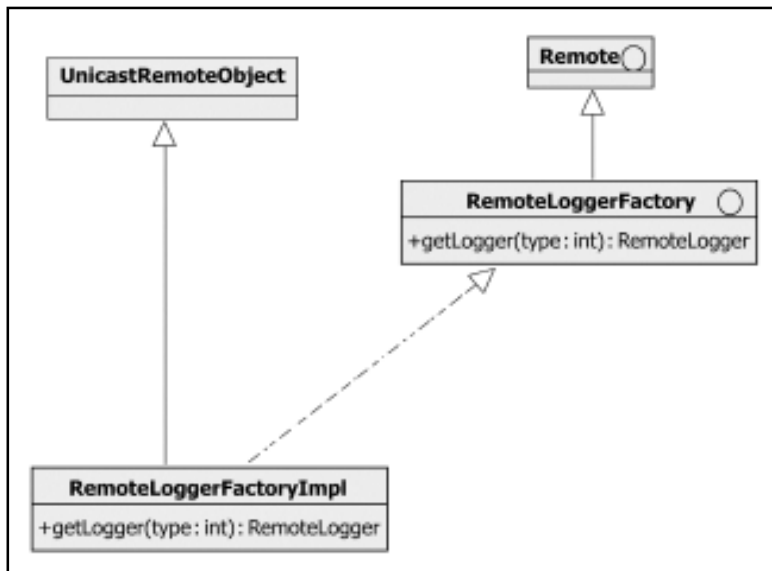
Come si può notare, la classe base è stata sostituita dalla interfaccia remota `RemoteLogger`, la quale, oltre a sopperire alla funzione di interfaccia standard, permette alle sottoclassi di essere oggetti remoti a tutti gli effetti.

Si è predisposta in tal modo la base per permettere l'utilizzo da client RMI di oggetti remoti. A questo punto si deve predisporre un qualche meccanismo che permetta la creazione di tali oggetti anche da parte di client RMI.

La classe `LoggerFactory` che implementa il pattern Factory, è a tutti gli effetti un oggetto remoto come quelli visti in precedenza: tramite la sua registrazione, ogni client potrà ottenerne lo stub ed invocarne il metodo remoto `getLogger()`, la cui implementazione è riportata di seguito.

```
public RemoteLogger getLogger(int type) throws RemoteException {
   RemoteLogger ret = null;

   if (type == 1) {
      ret = new RemoteFileLogger();
   }

   if (type == 2){
      ret = new RemoteStreamLogger();
   }
   return ret;
}
```

Tale metodo, controllando il valore del parametro passato, è in grado di restituire un oggetto (remoto) di tipo `FileLogger` e `StreamLogger`; più precisamente viene restituita un'interfaccia remota `Logger`, cosa che permette di mantenere una grande genericità.

Figura 6.6 – *Il factory remoto*

Da un punto di vista organizzativo le classi remote che implementano il Factory sono strutturate secondo lo schema UML riportato in fig. 6.6. Il client invocando il metodo remoto `getLogger()` riceverà lo stub dell'oggetto remoto, il quale verrà eseguito sul server remoto, in perfetto accordo con il modello teorico di RMI.

Attivazione di oggetti remoti

L'utilizzo del pattern Factory risolve in maniera piuttosto agile alcune delle limitazioni imposte dal modello RMI: con tale tecnica infatti è possibile attivare oggetti al momento della effettiva necessità su richiesta del client, e in modo esclusivo.

Rispetto alla situazione standard, il problema dello spreco delle risorse è quindi ridotto, dato che, tranne che per il factory remoto, gli altri oggetti remoti sono inizializzati al momento dell'effettivo bisogno.

Con il rilascio della piattaforma Java 2 è stata introdotta una tecnica alternativa, che si basa su un altro tipo di oggetti remoti: invece che derivare da `UnicastRemoteObject`, in questo caso l'oggetto attivabile su richiesta del cliente estende direttamente la classe `java.rmi.activation.Activatable`.

Questa maggiore flessibilità permette un più razionale utilizzo delle risorse, senza sconvolgere di fatto l'organizzazione delle applicazioni, dato che dal punto di vista del client RMI il meccanismo di invocazione remota è identico al caso standard.

Le operazioni da fare per modificare un oggetto remoto riguardano, come accennato in precedenza, esclusivamente il lato server. L'oggetto remoto in questo caso deve essere modificato nel seguente modo:

1. estendere la classe `Activatable` invece che `UnicastRemoteObject`

```
import java.rmi.*;
import java.rmi.activation.*;
public class Active extends Activatable
```

   si noti l'import del package `java.rmi.activation` al posto del `java.rmi.server`;

2. rimuovere o commentare il costruttore vuoto che prima era obbligatoriamente necessario definire; al suo posto mettere la versione riportata di seguito

```
public Active(ActivationId id, MarshallObject data) throws Exception {
   super(id, data);
}
```

A questo punto formalmente l'oggetto non è più un oggetto remoto ma è detto attivabile.

Per quanto riguarda l'applicazione server vi è un'altra importante differenza: nel caso degli oggetti remoti essa doveva restare attiva per tutto il tempo necessario all'utilizzo degli oggetti remoti da parte dei client. Adesso invece gli oggetti remoti sono gestiti da un demone apposito (`rmid`), ed il server deve esclusivamente effettuare l'operazione di registrazione, e dopo che tale installazione è terminata, il server può uscire. Per questo motivo in genere si utilizzano nomi del tipo *`setupXXX`* invece di *`serverXXX`*.

Il processo di installazione si traduce nel creare un ambiente di lavoro per l'oggetto attivabile (quello che si definisce `ActivationGroup`), settando eventualmente delle variabili che verranno utilizzate al momento della inizializzazione dell'oggetto attivabile; successivamente tale setup-class provvede a registrare tale oggetto nel registro RMI.

Di seguito sono riportate sinteticamente le operazioni da svolgere per installare un oggetto attivabile.

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

//installare un security manager appropriato

//creare un'istanza di ActivationGroup

Properties props;
props  = (Properties)System.getProperties().clone();
ActivationGroupDesc Agd;
```

```
Agd = new ActivationGroupDesc(props, null);
ActivationGroupID Agi;
Agi  = ActivationGroup.getSystem().registerGroup(Agd);
ActivationGroup.createGroup(Agi, Agd, 0);
```

Si deve infine creare una istanza di `ActivationDesc` la quale fornisce tutte le informazioni di cui il demone `rmid` necessita per creare le istanze vere e proprie degli oggetti attivabili.

Queste informazioni consistono di varie parti, come ad esempio il nome della classe, la collocazione del codice remoto, e una istanza della classe che serve per passare una configurazione di inizializzazione all'oggetto remoto.

Ad esempio si può scrivere

```
ActivationDesc Desc;
Desc  = new ActivationDesc("EchoApp.EchoImpl", location, data);
```

La classe `MarshalledObject`, introdotta con il JDK 1.2, contiene un bytestream dove viene memorizzata un rappresentazione serializzata dell'oggetto passato al suo costruttore.

```
public MarshalledObject(Object obj) throws IOException
```

Il metodo `get` restituisce una copia non serializzata dell'oggetto originale. La modalità con cui vengono effettuate le operazioni di serializzazione e deserializzazione sono le stesse utilizzate durante il processo di marshalling dei parametri durante una invocazione RMI di un metodo remoto.

Durante la serializzazione l'oggetto viene memorizzato con il codebase dell'URL da dove la classe eventualmente può essere scaricata. Inoltre ogni oggetto remoto memorizzato all'interno del `MarshalledObject` è rappresentato con una istanza serializzata dello stub dell'oggetto stesso.

La classe `MarshalledObject` è utile in tutti quei casi in cui si debbano implementare manualmente alcuni passaggi tipici di RMI, come il trasferimento di uno stub da client a server, oppure la serializzazione dello stesso secondo la semantica di serializzazione utilizzata in RMI.

Molto utile la possibilità di scaricare, per mezzo del metodo `get`, la classe corrispondente all'oggetto remoto, se questa non è presente in locale al momento della lookup da parte del client RMI. Questa funzione infatti permette di risolvere uno dei problemi principali di RMI, ovvero la necessità di dover installare sul client il codice (`.class`) corrispondente allo stub dell'oggetto remoto, prima di effettuare una lookup.

Bibliografia

[serialization] La specifica della serializzazione
ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf

[serialFAQ] FAQ sulla serializzazione
http://java.sun.com/products/jdk/serialization/faq

[rmi] Home page di RMI
http://java.sun.com/products/jdk/rmi/index.html

[rmispec] La specifica di RMI
ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf

[GOF] Erich Gamm – Richard Helm – Ralph Johnson – John Vlissides, *Design Pattern*, Ed. Addison Wesley

[java pattern] James W. Cooper, *Java Design Pattern, A tutorial*, Ed. Addison Wesley

[mokabyte pattern] *Corso pratico di pattern*, in MokaByte n. 26 (gennaio 1999) e successivi
www.mokabyte.it/1999/01

Java e i database

DI NICOLA VENDITTI

Che cosa sono i database?

Ogni giorno nel mondo vengono scambiati inimmaginabili volumi di dati e quasi in ogni parte del mondo le informazioni vengono recuperate, elaborate, trasformate, accresciute, aggiornate e infine re-immagazzinate. Questa enormità, che è una novità della nostra epoca, rappresenta, per la sua portata, un raggiungimento per il genere umano.

I databases sono il centro vitale di questo movimento: ci permettono non solamente di archiviare i dati e le informazioni raccolte nei vari campi dell'attività economica, scientifica ecc., ma garantiscono anche la sicurezza e l'integrità dei dati medesimi, così come la possibilità di poterli recuperare in ogni momento nel modo più efficiente e rapido possibile.

A mano a mano che l'informatica ha conquistato un ruolo fondamentale nei vari campi dell'attività umana, è emersa la necessità di avere dei sistemi capaci di gestire in modo avanzato i dati e le informazioni. I DBMS (DataBase Management System) sono la risposta più significativa a questa esigenza. Per DBMS si intende un sistema costituito essenzialmente dal database vero e proprio e, soprattutto, dal software per gestire tutte le operazioni che ordinariamente si fanno su un database, dall'archiviazione all'aggiornamento, fino al backup, al mirroring e così via.

Risale agli inizi degli anni Ottanta la comparsa sul mercato software dei primi DBMS e si trattava per lo più di sistemi che usavano il file system del sistema operativo che li ospitava come repository dei dati e delle librerie C per accedere ad essi da parte dei programmi client.

Pioniere nel campo della ricerca orientata ai database è stata la IBM che, prima ancora di altri grandi database vendor come Oracle e Informix, si trovò ad affrontare la necessità di fornire sui propri sistemi mainframe, allora già largamente diffusi, del software capace di gestire l'archiviazione dei dati.

Come detto, originariamente, e in parte anche adesso, era il file che veniva utilizzato come unità di storage per i database . Così la ricerca si orientò allo studio di un metodo di organizzazione e strutturazione dello spazio nei files per un archiviazione ottimale e un accesso efficiente ai dati. Un risultato tuttora popolare di questa ricerca fu l'ISAM (Indexed Sequential Access Method).

Il concetto di tabella divenne popolare insieme al modello relazionale agli inizi degli anni Settanta grazie a Codd (un ricercatore di IBM), che gettò le basi di un approccio teorico ancora largamente utilizzato in questo settore.

Con la comparsa di nuovi protagonisti nel campo dei database, sorse l'esigenza di avere un linguaggio comune per l'accesso ai dati, visto che ognuno disponeva di una propria libreria (ad esempio Informix nelle primissime versioni del suo database forniva una libreria detta C-ISAM).

Anche in questo la IBM fu protagonista, e finì per diffondersi un suo linguaggio chiamato SQL (Structured Query Language) oggi molto popolare: da allora non ha subito modifiche ma solo aggiunte.

L'SQL fu derivato a sua volta da un altro linguaggio sperimentale chiamato SEQUEL (da qui la diffusa pronuncia "siquel") creato per un sistema che si chiamava System R. La standardizzazione dell'SQL voluta da ISO e ANSI risale al 1986. Una successiva standardizzazione, nel 1992, introduce nuovi e interessanti elementi senza cambiare la struttura. A questa versione dell'SQL si ci riferisce come SQL-92. L'ultima versione di SQL è SQL99 ovvero SQL3 che introduce il supporto per gli oggetti.

Prima di passare al punto di vista client, e quindi a Java e JDBC, ecco una breve panoramica dei tipi di database esistenti sul mercato. Il modo più comune per classificare i database è quello di discriminarli in base al modello di organizzazione dei dati che utilizzano al loro interno. Usando questo metodo una possibile suddivisione dei databases potrebbe essere quella che segue nei prossimi paragrafi.

Relazionali

Si basano sul modello relazionale: prevedono quindi organizzazione dei dati concettualmente descrivibile in termini di entità e relazioni tra di loro; l'accesso e la manipolazione dei dati viene fatto tramite SQL.

Esempi: Oracle 8i, Informix Dynamic Server.2000, DB2 e altri.

Dimensionali

Sono una specializzazione dei primi per il datawarehouse: utilizzano il modello relazionale ma i princìpi con cui viene definito il modello di un DB dimensionale è diverso da quello tradizionale, basato sul concetto di normalizzazione. Per accedere e utilizzare le caratteristiche di questi database si utilizza una versione estesa dell'SQL. Questi database vengo-

no più comunemente chiamati OLAP (On Line Analytical Processing) per confronto con i precedenti anche conosciuti con il nome di OLTP (On Line Transaction Processing).

Esempi: Redbrick, Informix XPS e altri.

Object Oriented

Sono molto più recenti dei primi due. Più che basi di dati sono framework per la persistenza di oggetti applicativi. I linguaggi a disposizione per questi database, riconosciuti e standardizzati da OMG (Object Management Group) sono due: OQL (Object Query Language) per la manipolazione dei dati e ODL (Object Definiction Language) per la definizione dello schema.

Esempi: Jasmine di CA.

Object Relational

Rappresentano una via di mezzo tra i database relazionali puri e i database OO anche se più esattamente possono essere considerati dei relazionali con estensioni di supporto per la tecnologia Object Oriented.

Per accedere a questi database si utilizza ancora l'SQL, ed è lo stesso SQL, attraverso alcune estensioni proprietarie, a permettere l'accesso alle caratteristiche a oggetti di questi database.

Accesso ai database: il punto di vista applicativo

Parallelamente all'evoluzione dei database sono cambiati i meccanismi con cui le applicazioni accedono ai dati. Originariamente come detto ogni DBMS disponeva, per le applicazioni, di proprie librerie C. Oltre all'SQL fu creato uno standard a "livello di chiamata" detto appunto Call Level Interface (CLI) proposto da X/Open. Fu cioè definita la sequenza di chiamate che l'applicazione (per lo più un'applicazione C) doveva seguire per accedere in modo corretto alle informazioni. I produttori di database hanno iniziato a fornire questo set di librerie in aggiunta alle proprie originarie. Per esempio ancora adesso il database Oracle fornisce uno strato CLI detto OCI (Oracle Call-level Interface).

JDBC è modello anch'esso basato sullo standard CLI per l'accesso alle basi di dati. Il gruppo responsabile di Java ha deciso, a ragione, di fornire il proprio linguaggio di una propria libreria per le operazioni sui dati. Le motivazioni sono diverse ma si capisce subito che una libreria scritta per il C difficilmente si adatta a un linguaggio basato su classi e interfacce (e, per giunta, a differenza del C++, privo di puntatori).

Il risultato raggiunto brilla sicuramente per semplicità ed essenzialità: con solamente 4 righe di codice posso caricare il driver più adatto alla base di dati che voglio interrogare, ottenere la connessione, creare lo statement e recuperare il result set esplicitando la query.

I problemi che occorre affrontare quando si scrive una libreria per l'accesso ai database sono diversi e non solo di tipo applicativo: si deve garantire dal lato client una coerenza logica il più possibile vicina alla filosofia del linguaggio che l'applicazione usa, adattandosi ai metodi tra loro molto differenti che i DBMS utilizzano per processare le richieste dei client; si deve fornire poi una specifica per la scrittura e l'implementazione dei drivers e non da ultimo convincere i produttori di database della opportunità di scrivere drivers per questa nuova interfaccia. Pertanto il risultato di semplicità e universalità di JDBC è tanto più apprezzabile.

JDBC è una interfaccia a oggetti per l'esecuzione di comandi SQL: è bene sottolineare quindi che il vero medium per la comunicazione con il database rimane il linguaggio SQL e lo stesso vale per le altre tecnologie a oggetti concorrenti di Java come DAO di Microsoft.

L'accesso ai dati introduce un'anomalia nel paradigma Object Oriented, dove non esiste il concetto di dato semplice né tantomeno quello di riga e di tabella. Per questi linguaggi occorrerebbe, più che una repository di dati come i database tradizionali, un framework per la persistenza degli oggetti: tutto quello che l'applicazione dovrebbe fare sarebbe solo di indicare un'astratta repository da cui recuperare un oggetto che rappresenta un certo elemento di realtà e in cui immagazzinare o più precisamente rendere persistenti gli oggetti nel loro stato applicativo.

A parte questo problema, JDBC è molto flessibile, anche grazie al fatto che è stato progettato basandosi sul concetto di interfaccia e non di oggetto: tutte le operazioni necessarie per accedere ai dati vengono fatte attraverso i metodi esposti da interfacce (che il driver JDBC si occupa di implementare) e non occorre istanziare alcun oggetto.

Introduzione a JDBC

Spiegare come è fatto JDBC significa essenzialmente dire come una applicazione deve utilizzarlo, perché in fondo non è altro che una libreria per applicazioni Java. Per accedere alla funzionalità di JDBC occorre che l'applicazione importi il package `java.sql`, parte integrante di tutte le Virtual Machine a partire dalla versione 1.1.

Al momento la versione più recente di JDBC è la 2.1, che comprende tutto il JDBC 1.1 e lo estende con nuove funzionalità e in più l'Optional Package, per l'integrazione di JDBC nella piattaforma J2EE; si vedranno più avanti le novità introdotte con JDBC 2.1 e cos'è l'Optional Package (ex Standard Extentions). La versione 3.0 di JDBC sarà parte integrante di Java Standard Edition v1.4 che includerà tanto il classico package `java.sql`, che l'Optional Package `javax.sql`.

Prima di vedere, con l'ausilio di alcuni esempi, come funzionano le classi JDBC, vediamo brevemente una tipica applicazione. Si supponga di disporre di un DB per il quale esista un driver JDBC. I passi da compiere per l'esecuzione di una semplice query sono principalmente i seguenti:

1. Caricamento driver JDBC e connessione al DB

Prima di ogni cosa occorre caricare il driver che gestisce la nostra base dati, in modo esplicito o implicito. Ecco un esempio di caricamento esplicito della classe di un driver:

```
Class.forName("com.informix.jdbc.IfxDriver");
```

per caricare il driver JDBC tipo 4 di Informix.

La seconda operazione da fare è la connessione al DB, univocamente individuato dalla stringa o URL di connessione. Al termine di questa operazione si dispone di un oggetto di tipo `Connection` che rappresenta la connessione.

Per la connessione l'applicazione si affida al Driver Manager; l'applicazione semplicemente richiede una connessione specificando l'URL del database a cui desidera connettersi.

```
Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@ORCL:1025", "scott", "tiger");
```

Sarà il Driver Manager a recuperare il driver giusto attraverso un meccanismo a cui devono conformarsi tutti i driver JDBC. Una volta selezionato, il driver sarà responsabile di tutte le operazioni per quella connessione.

2. Creazione dello Statement object

Creata la connessione al DB si è in possesso di un oggetto che rappresenta la connessione al DB. Da esso è possibile poi ottenere tre diversi tipi di oggetti che permettono di creare istruzioni SQL da inviare al DB. Questi sono `Statement`, `PreparedStatement` e `CallableStatement`. Alcuni DBMS potrebbero non avere tutte e tre le implementazioni dal punto di vista nativo. Ecco il frammento di codice che recupera lo statement

```
Statement stmt = conn.createStatement();
```

Come vedremo, si possono recuperare versioni più specializzate per lavorare per esempio con le stored procedure; l'esempio sopra riportato è però sufficiente a dare l'idea.

3. Esecuzione della query

Una volta creato uno dei tre oggetti precedenti non ci resta altro che eseguire la query. Le specifiche JDBC vogliono che le operazioni avvengano in modo *thread safe* in maniera tale che oggetti diversi accedano agli stessi dati in modo sicuro.

Ecco un esempio

```
ResultSet rs = stmt.executeQuery("SELECT * FROM utenti");
```

4. Elaborazione dei risultati

Come si vedrà una volta eseguite le query si ottiene un oggetto di tipo `ResultSet` che contiene i dati risultati della query ordinati per record. Attraverso il suo metodo `next()` è possibile percorrere tali record e accedere ai campi dello stesso attraverso opportuni metodi `getXXX`.

Come detto all'inizio, quando si costruisce una libreria per i database occorre che questa si conformi al modello di accesso ai dati utilizzato del medesimo. Questo è un esempio evidente. In questo caso sembra di avere a disposizione un oggetto che contiene al suo interno il set di dati recuperato ma non è così. In realtà la query ha solo creato un cursore che punta inizialmente alla prima riga del set di risultati ritornati: il metodo `next()` serve per spostare il cursore in avanti e i metodi `getXXX()` si utilizzano per spostare effettivamente i dati dal server al client.

Per rendersene conto è possbile utilizzare il seguente drastico esperimento: si provi a eseguire i passi fino al 4 e, dopo la prima chiamata a `next()`, si tenti di distruggere la tabella su cui è stata fatta la query. Se non vi fossero i cursori e l'applicazione avesse tutti i risultati della query, allora potrebbe scandire il suo result set anche se la tabella originaria non esiste più, e invece viene segnalato un errore grave. Si tornerà su questa differenza a proposito dei `RowSets`.

Il metodo standard per la scansione dei risultati è mostrato nel codice che segue.

```
while(rs.next()) {
...
String nome = rs.getString(1);    // colonna 1 contiene il nome
String cognome = rs.getString(2); // colonna 2 contiene il cognome
int eta = rs.getInt(2);           // colonna 3 contiene l'età
...
}
```

5. Eventuale commit e rollback

JDBC offre la possibilità di gestione delle transazioni. Per inciso, il database a cui si connette l'applicazione deve supportare le transazioni. Ciò si traduce nel fatto che nel database dell'applicazione deve essere abilitato il log delle operazioni. Verificare presso l'amministratore del DB che sia effettivamente così.

Se il logging sul database non è abilitato, e quindi non sono supportate le transazioni, si riceve un esplicito messaggio di errore in tal senso. La commit o la rollback della transazione vanno invocate attraverso gli omonimi metodi dell'oggetto `Connection`.

6. Rilascio delle risorse utilizzate

Una volta eseguite le operazioni volute è bene rilasciare le risorse acquisite per cui si

chiude la connessione al DBMS. Ciò comporta per lo meno la chiusura degli oggetti `ResultSet`, `Statement` e `Connection`, nell'ordine inverso alla loro apertura.

Se occorre, è disponibile un metodo su tali oggetti per verificare se l'operazione è già stata effettuata: `isClose()`, che però vi dice solo che il metodo `close()` sull'oggetto è stato già chiamato, e non se la connessione è attiva o meno.

Ecco tradotta in un piccolo esempio di programma che utilizza JDBC su ODBC, la sequenza di passi appena illustrata.

```
try {
   // Caricamento esplicito del driver JDBC per il tipo di sorgente
   // di dati che mi interessa: in questo caso una fonte ODBC
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

   // Creazione della connessione al database (tramite
   // il Driver Manager)
   Connection con = DriverManager.getConnection("jdbc:odbc:MokaDb");

   // Creazione dello statement, l'oggetto che utilizzo per
   // spedire i comandi SQL al database
   Statement st = con.createStatement();

   // Eseguo la query o un altro comando SQL attraverso lo statement
   // e recupero i risultati attraverso l'interfaccia ResultSet
   ResultSet rs = st.executeQuery("select * from lettori");

   // Scandisco il result set per la visualizzazione
   // delle informazioni recuperate
   while (rs.next()){
      System.out.println("Nome: " + rs.getString(1));
      System.out.println("Cognome: " + rs.getString(2));
      System.out.println("Telefono: " + rs.getString(3));
      System.out.println("Email: " + rs.getString(4));
      System.out.println("Categoria: " + rs.getString(5));
      System.out.println(" ");
   }

   // Rilascio le risorse utilizzate
   rs.close();   // result set
   st.close();   // statement
   con.close();  // connessione
}
catch(ClassNotFoundException cnfe) {
   System.err.println("Attenzione classe non trovata" + cnfe.getMessage());
}
catch(SQLException sqle) {
   System.err.println("Attenzione errore SQL" + sqle.getMessage());
}
```

Fino ad ora è stata mostrata la sequenza di passi da seguire per l'esecuzione di una query. Ovviamente JDBC permette di eseguire tutte le operazioni SQL di manipolazione dei dati e delle strutture dati (comandi DDL e DML).

Convenzioni JDBC URL Naming

Per permettere a una applicazione di accedere a un database, la prima cosa da fare è definire la localizzazione dello stesso e la modalità di accesso, come ad esempio il tipo di driver da utilizzare oppure UserID e Password dell'utente con il quale si intende accedere ai dati. Tutte queste informazioni devono essere rese disponibili al momento della connessione in un qualche formato prestabilito. La soluzione scelta consiste nell'impacchettare tutti questi dati in una stringa, detta di connessione, che viene utilizzata al momento della creazione del collegamento. Per la scelta della sintassi di tale stringa si è adottato un formalismo derivante da quello utilizzato per la definizione degli URL.

La struttura generica di tale stringa è la seguente:

```
jdbc:<subprotocol><domain-name>
```

Nella sintassi URL il primo termine indica il protocollo da utilizzare per la gestione della risorsa individuata, e che in questo è ovviamente `jdbc`.

Il subprotocol rappresenta invece il sottoprotocollo ovvero il driver che si intende utilizzare come interfaccia al DB verso il DBMS. Per domain-name si intende, invece, il nome della risorsa che verrà elaborata dal driver in relazione a delle regole caratteristiche del driver stesso. Nel caso più comune di utilizzo del bridge `jdbc-odbc`, un URL potrebbe essere il seguente:

```
jdbc:odbc://www.infomedia.it/dblettori
```

In questo modo si indica che la risorsa che vogliamo raggiungere attraverso JDBC e il driver bridged JDBC-ODBC, si chiama `dblettori`. Come detto l'interpretazione del domain-name dipende dal driver. Nel caso del bridge il nome `dblettori` rappresenta il nome di un DSN (Data Source Name). Nel seguente caso

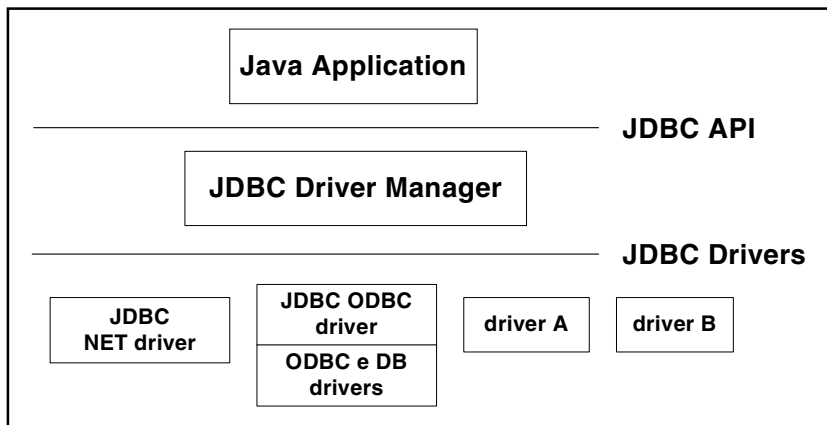
```
jdbc:infonaming:DB
```

si interpreterà il nome `DB` in relazione a quello stabilito dal sottoprotocollo `infonaming` che potrebbe, ad esempio, indirizzare al DBMS in un URL caratteristico dello stesso.

Gestione dei driver: il DriverManager

Come si può dedurre dalle considerazioni fatte, i driver assumono un ruolo fondamentale. Ad essi è delegato il compito di adattare tutte le operazioni possibili con JDBC al DBMS corrispondente. JDBC dà la possibilità di collegarsi a diversi DB individuati da

diversi URL. Serve quindi un meccanismo per associare a ciascun sottoprotocollo il driver corrispondente. A questo compito è dedicata la classe `java.sql.DriverManager`. Essa gestisce i driver e permette di stabilire una connessione con il DBMS indirizzato fornendo un oggetto di tipo `java.sql.Connection` rappresentativo della connessione.

Figura 7.1 – *Organizzazione funzionale delle varie componenti del package* `java.sql`

Il meccanismo di caricamento del driver corrispondente all'URL può avvenire in due modi:

- attraverso la lista dei driver elencati nella proprietà di sistema `jdbc.drivers`. I vari driver disponibili saranno elencati nella proprietà `jdbc.drivers`, in successione divisi dai due punti. Quando si cerca, attraverso il metodo `getConnection()` di stabilire una connessione al DB e quindi di ottenere un oggetto `Connection`, il `DriverManager` carica tutti i driver elencati nella proprietà di sistema `jdbc.drivers`. Poi registrerà i driver trovati attraverso il metodo `registerDriver()`. Per stabilire la connessione, il `DriverManager` farà un parsing dell'URL e cercherà il driver corrispondente tra quelli memorizzati. Possiamo ottenere la lista di questi attraverso il metodo `getDrivers()`.

- richiamando esplicitamente il driver attraverso la istruzione `Class.forName()`. In questo caso si avrà il caricamento del driver e la gestione dello stesso come se fosse avvenuta nel modo descritto al punto precedente.

Nell'esempio che segue è riportata una porzione di codice completa che partendo dalle due modalità di caricamento del driver, esegue alcune semplici operazioni, come la connessione e il successivo controllo sui driver disponibili.

```
...
if (loadtype.compareTo("Esp")==0) {
   // Caricamento del driver per mezzo del settaggio
   // della proprietà di sistema sql.drivers
   System.out.println("Caricamento implicito del Bridge JDBC-ODBC");
   Properties prop = System.getProperties();
   prop.put("jdbc.drivers", "sun.jdbc.odbc.JdbcOdbcDriver");
   System.setProperties(prop);
   }
   else {
      //caricamento del driver per dichiazione esplicita
      try {
         System.out.println("Caricamento esplicito di JDBC-ODBC");
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }
      catch(ClassNotFoundException cnfe) {
         System.out.println("Attenzione: driver non disponibile");
      }
   }

   // controllo di tutti i driver caricati
   Enumeration enum = DriverManager.getDrivers();
   System.out.println("");
   System.out.println("Elenco dei driver disponibili in memoria");
   System.out.println("————————————————————");
   System.out.println("Driver =" + enum.nextElement());
      while (enum.hasMoreElements())
         System.out.println("Driver" + enum.nextElement());
   }
...
```

La classe `DriverManager` non è l'unica che permette di stabilire una connessione al DB. Esiste anche l'implementazione dell'interfaccia `Driver`. Essa contiene tutte le informazioni relative a un particolare driver. È possibile conoscere se esso è JDBC-Compliant, la sua versione, se accetta oppure no un determinato URL. Attraverso il metodo `connect()` è possibile creare la connessione a un URL. Questa interfaccia è utile nel caso di gestione avanzata di determinati `Driver` in quanto permette, attraverso un oggetto `Properties` di assegnare valori a suoi eventuali parametri. A tale scopo è dedicata anche la classe `DriverPropertyInfo`.

Ecco un esempio di utilizzazione dell'interfaccia `Driver` per conoscere le caratteristiche del database.

```
...
try {
   // caricamento del driver per dichiazione esplicita
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(ClassNotFoundException cnfe) {
   System.out.println("Attenzione il driver non è disponibile");
}
try {
   Enumeration drivers = DriverManager.getDrivers();
       while (drivers.hasMoreElements()) {
          Driver driver = (Driver) (drivers.nextElement());
       System.out.println("Driver: " + driver);
       System.out.println("Accetta standard jdbc-URL: "
                          + driver.acceptsURL("jdbc:odbc:MokaDb"));
          System.out.println("Minor Version: " + driver.getMinorVersion());
       System.out.println("Major Version: " + driver.getMajorVersion());
          System.out.println("JDBC Compatibile:" + driver.jdbcCompliant());
   }
}
```

Gestione degli errori

Si prende ora in considerazione un aspetto abbastanza importante, quello della gestione delle eccezioni durante questo tipo di operazioni.

Le SQLExceptions

Negli esempi riportati precedentemente si è potuto osservare che, ogni volta che viene eseguita una operazione delicata, può essere intercettata una eccezione di tipo `SQLException`. Questa classe offre una serie di informazioni relativamente al tipo di errore verificatosi. Essa deriva dalla più generale `java.lang.Exception`, la quale a sua volta deriva dalla `Throwable`. Le informazioni contenute nella classe sono le seguenti:

- il tipo di errore verificato sotto forma di una stringa descrittiva; tale informazione può essere utilizzato come Exception message e può essere ricavata per mezzo del metodo `getMessage()`.

- una proprietà (`SQLState`) descrivente l'errore in base alle convenzioni dello standard X/Open SQLState. Può essere ottenuto con `getSQLState()`.

- un codice di errore specifico del produttore del database, che in genere corrisponde al messaggio di errore fornito dal DBMS stesso; `getErrorCode()` permette la sua lettura.

- una catena al successivo oggetto di tipo SQLException, la quale può essere utilizzata se si sono verificati più di un errore. Il metodo getNextException() permette di spostarsi su questa catena.

Nell'esempio che segue è mostrato come utilizzare tali informazioni per avere una descrizione completa dell'errore verificatosi.

```
...
try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection con = DriverManager.getConnection("jdbc:odbc:MokaDb");
   Statement st = con.createStatement();
   //esegue una istruzione errata: la tabella Lettrici non esiste
   ResultSet rs = st.executeQuery("select * from Lettrici");
}
catch(ClassNotFoundException cnfe) {
   System.out.println("Classe non trovata" + cnfe.getMessage());
}
catch(SQLException sqle) {
   System.out.println("Attenzione errore SQL" + "\n");
   while (sqle != null) {
      System.out.println("Messaggio SQL \n" + sqle.getMessage() + "\n");
      System.out.println("SQLState \n" + sqle.getSQLState() + "\n");
      System.out.println("Codice errore del produttore \n"
                         + sqle.getErrorCode()+ "\n");
      System.out.println("Traccia dello StackTrace");
      sqle.printStackTrace(System.out);
      sqle = sqle.getNextException();
      System.out.println("");
   }
}
```

I Warnings

Oltre alle eccezioni alle quali si è abituati in Java, nel caso specifico di JDBC è disponibile anche un altro strumento per la gestione delle situazioni anomale. La differenza che sussiste fra un Warning e una eccezione è essenzialmente nel fatto che il primo non interrompe il flusso del programma: si dice infatti che sono dei segnalatori silenziosi di anomalie. Un Warning viene scatenato direttamente dal database, in funzione del tipo di errore che si verifica.

Non è possibile generalizzare quando l'uno o l'altro tipo di strumento viene generato, essendo molto dipendente dall'implementazione del DB. I Warning offrono lo stesso tipo di informazioni delle eccezioni SQLException, e possono essere ricavati con metodi del tutto identici rispetto al caso precedente.

I metadati

JDBC permette quello che la documentazione di Sun chiama "accesso dinamico ai database". Il nome non indica molto, ma si tratta della possibilità di accedere a un database e ricavarne informazioni sulla sua struttura interna (tabelle, relazioni, sinonimi, link, trigger, ecc.) senza saperne nulla a priori. In questo modo posso, per esempio, interrogare una tabella utenti senza sapere in anticipo quali e quante sono le colonne che la compongono.

La risposta è nel fatto che tutti i DBMS hanno delle tabelle interne dette "dizionari" o "cataloghi" che contengono metainformazioni circa la struttura interna dai database. Per fare un esempio, quando si crea un database Informix, prima ancora che vengano create tabelle dagli utenti abilitati, ne viene automaticamente creata qualche decina come normale processo di creazione del database. Se si hanno i privilegi e si esegue il comando `"SELECT * FROM systables"` si può aver l'elenco di tali tabelle che hanno nomi a volte molto espliciti (`sysindexes, systriggers, syscolumns`, ecc.).

JDBC sfrutta questa possibilità e fornisce un paio di interfacce per fornire al programmatore un modo per accedere a tali meta-informazioni.

Si conclude questa panoramica su JDBC con una applicazione completa che può essere utilizzata per eseguire query su database Cloudscape. Cloudscape è il database della omonima software house recentemente aquisita da Informix, distribuito in evaluation dalla Sun con la propria piattaforma Java 2 Platform Enterprise Edition. Per provare il programma quindi si può utilizzare l'installazione di J2EE presente sul computer oppure scaricare una versione gratuita di Cloudscape per valutazione.

Cloudscape può anche essere utilizzato come database di test per le tecnologie J2EE che agiscono su Data Sources, pertanto è comodo un programma da linea di comando per l'esecuzione di queries come il seguente.

```
package esempi.cloudscape;

import java.sql.*;

/**
 * Semplice programma che crea e si connette a
 * un database <i>Cloudscape</i>
 */
public class SQLCloudscape {

/**
 * Nome del driver JDBC di Cloudscape
 * Una volta installata la Java 2 EE si trova in
 * $J2EE_HOME/lib/cloudscape/cloudscape.jar su Unix e
 * %J2EE_HOME%\lib\cloudscape\cloudscape.jar su Windows
 */
static String driver = "COM.cloudscape.core.JDBCDriver";
```

```
/**
 * Stringa di connessione al database cloudscape
 * 'create = true' indica al driver di creare il database
 * se non esiste. Viene utilizzato se non ne viene fornito
 * uno dalla linea di comando
 */
static String url = "jdbc:cloudscape:clouddb;create = true";

/**
 * Utilizzato come comando SQL di default se non ne viene
 * passato uno sulla linea di comando
 */
static String sql = "SELECT * FROM SYS.SYSTABLES";

/**
 * main: entry del programma
 */
public static void main(String[] args) {
   try {

      if(args.length > 0)
         url = args[0];

      if(args.length > 1)
         sql = args[1];

      // Carica il driver di Cloudscape
      Class c = Class.forName(driver);
      System.err.println("Caricato driver nella VM: " + c.toString());

      // Crea la connessione (crea il database)
      Connection conn = DriverManager.getConnection(url);
      System.err.println("Creata connessione: " + conn.toString());

      // Crea lo statement
      Statement stmt = conn.createStatement();
      System.err.println("Creato statement: " + stmt.toString());

      // Esegue una query sulla tabella di sistema
      ResultSet rs = stmt.executeQuery(sql);
      System.err.println("Esecuzione query: " + sql);

      // La scansione result set dà tutte le tabelle
      // presenti nel database
      ResultSetMetaData rsmd = rs.getMetaData();
      // prende il numero di colonne
      int cols = rsmd.getColumnCount();

      while(rs.next()) {
```

```
         for(int i = 0; i < cols; ++i) {
            // le colonne 1, 2
            System.out.print(rs.getString(i + 1));
            // separatore di campi
            if(i < cols) System.out.print("|");
         }
         System.out.println("");
      }

      // Chiusura delle risorse
      rs.close();
      stmt.close();
      conn.close();

   } catch(Exception e) {
         System.err.println("Messaggio err.: " + e.getMessage());
      }

      return;
   }

}
```

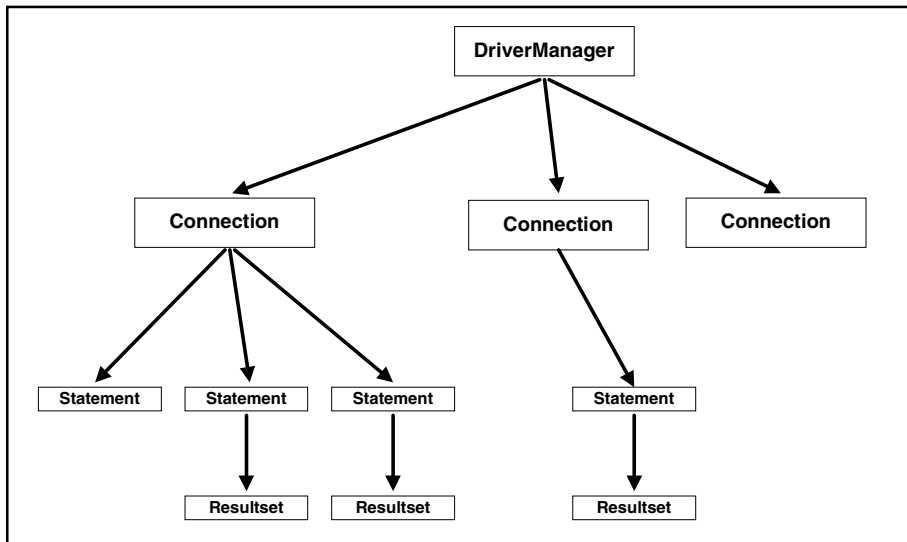
Segue lo script `sqlcloudscape.bat` che lancia il programma impostando correttamente l'ambiente.

```
REM
REM Lancia programma SQLCloudscape per le query su database Cloudscape
REM

SET J2EE_HOME=D:\java\j2sdkee1.2
SET CLOUDSCAPE_HOME=D:\java\j2sdkee1.2\cloudscape
SET WORK_HOME=D:\lavoro\sviluppo
SET CLASSPATH=%CLASSPATH%;%J2EE_HOME%\lib\cloudscape\cloudscape.jar;%WORK_HOME%


java -Dcloudscape.system.home=%CLOUDSCAPE_HOME% esempi.cloudscape.SQLCloudscape %1 %2
```

La variabile d'ambiente `cloudscape.system.home` indica dove creare il database se non specificato, come nel nostro caso.

JDBC 1.1 core API: struttura del package `java.sql`

Nella successiva fig. 7.2 è mostrato il modo in cui si articolano le classi più utilizzate del package `java.sql`.

Figura 7.2 – *Articolazione delle classi più utilizzate nel package* `java.sql`

In questa sezione si affrontano in dettaglio le principali classi del package `java.sql`.

Classe `DriverManager`

```
DriverManager.registerDriver()
```

Questo metodo restituisce un'enumerazione dei drivers caricati nella Virtual Machine. Per caricare un driver su utilizza di solito l'istruzione

```
Class.forName("com.informix.jdbc.IfxDriver");
```

I Driver JDBC hanno normalmente degli inizializzatori statici che registrano la classe stessa presso il DriverManager automaticamente. Ecco un esempio:

```
public class IfxDriver extends ... {
   // blocco di inizializzazione statica
   static {
   ...
   // qualcosa del genere...
   DriverManager.register(new IfxDriver());
   ...
```

```
   }
   ...
}
```

Questo metodo di registrazione viene detto implicito. Per registrare invece la classe del driver in modo esplicito si invoca il metodo `registerDriver()`.

```
DriverManager.registerDriver(new Classe_del_Driver);
```

Il thin driver di Oracle richiede la registrazione esplicita ad esempio, anche se il metodo più diffuso è quello implicito.

È possibile, come detto, fare in modo che la VM carichi automaticamente uno o più drivers JDBC. Per fare ciò occorre impostare la variabile d'ambiente `jdbc.drivers`.

```
DriverManager.getDrivers()
```

Questo metodo permette di recuperare in forma di enumerazione l'elenco dei drivers caricati nella VM. Il programma che segue utilizza `getDrivers()` per mostrare come effettivamente venga letta dalla VM la variabile d'ambiente `jdbc.drivers` citata poco sopra.

```
package esempi.sql;

import java.sql.DriverManager;
import java.sql.Driver;
import java.util.Enumeration;

/**
 * Esempio 01 - mostra la JVM
 * utilizza la variabile d'ambiente jdbc.drivers.
 */
public class Esempio01
{

   static public void main(String[] args) {
      // Mostra la variabile jdbc.drivers
      System.out.println("jdbc.drivers=" + System.getProperty("jdbc.drivers"));

      // Enumera i driver in memoria
      Enumeration e = DriverManager.getDrivers();
      while(e.hasMoreElements()) {
         Driver driver = (Driver)e.nextElement();
         System.out.println("Nome: " + driver.toString());
```

```
            System.out.println("JDBC Compliant: " + driver.jdbcCompliant());
         }
      }
   }
```

Ecco il risultato dell'esecuzione del programma in due diversi casi.

```
D:\lavoro\sviluppo>java esempi.sql.Esempio01
jdbc.drivers=null

D:\lavoro\sviluppo>java -Djdbc.drivers
=com.informix.jdbc.IfxDriver esempi.sql.Esempio01
jdbc.drivers=com.informix.jdbc.IfxDriver
Nome: com.informix.jdbc.IfxDriver@f87ee844
JDBC Compliant: true
D:\lavoro\sviluppo>java -Djdbc.drivers
=com.informix.jdbc.IfxDriver:COM.cloudscape.core.JDBCDriver esempi.sql.Esempio01
jdbc.drivers=com.informix.jdbc.IfxDriver:COM.cloudscape.core.JDBCDriver
Nome: com.informix.jdbc.IfxDriver@faf36c3c
JDBC Compliant: true
Nome: COM.jbms._46._217@86b76c3c
JDBC Compliant: false
```

`setLogStream()`

Questo metodo permette di impostare lo stream di log per le operazioni di JDBC. L'applicazione che segue mostra come si utilizza il metodo impostando come stream di log un file `sql.log`.

```
package esempi.sql;

import java.io.*;
import java.sql.*;

/**
 * Esempio 02 - mostra strumenti di loggin di JDBC
 */
public class Esempio02 {
   /**
    * Stream di log di JDBC
    */
   static PrintStream sqlstream = null;

   static {
```

```
      try {
            FileOutputStream fos = new FileOutputStream("C:\\temp\\sql.log");
         sqlstream = new PrintStream(fos);
         } catch(IOException e) {
         System.err.println("ERRORE I/O: " + e.getMessage());
      }
   }

   static public void main(String[] args) {
      try {

         // Carica il driver Cloudscape
         Class.forName("COM.cloudscape.core.JDBCDriver");

         DriverManager.setLogStream(sqlstream);
         DriverManager.println("— Sessione "
                              + (new java.util.Date()).toString() + " —");

         // Crea connessione
         DriverManager.println("apertura connessione...");
         Connection conn
         = DriverManager.getConnection("jdbc:cloudscape:clouddb");
         DriverManager.println("...connessione aperta con successo");

         // Chiude la connessione
         DriverManager.println("chiusura connessione...");
         conn.close();
         DriverManager.println("...connessione chiusa con successo");

      } catch(Exception e) {
         System.err.println("ERRORE: " + e.getMessage());
      }
   }
}
```

L'output prodotto nel file `C:\temp\sql.log` è il seguente:

```
    — Sessione Mon Oct 16 01:05:40 GMT+02:00 2000 —-
    apertura connessione...
DriverManager.getConnection("jdbc:cloudscape:clouddb")
    trying driver[className = COM.jbms._46._217,COM.jbms._46._217@8c4b62fb]
    getConnection returning
    driver[className=COM.jbms._46._217,COM.jbms._46._217@8c4b62fb]
    ...connessione aperta con successo
    chiusura connessione...
    ...connessione chiusa con successo
```

Si notano, oltre ai nostri messaggi, alcuni messaggi di log della VM riguardanti la ricerca del driver associato all'URL `jdbc:cloudscape:clouddb` utilizzato dal programma: questi messaggi non appaiono sulla console, quindi occorre esplicitamente impostare uno stream di log come mostrato (può anche essere `System.out` o `System.err`) per visualizzarli.

`getLoginTimeout()`

Il programma che segue recupera il default (`0`) per il valore di login timeout.

```
package esempi.sql;
import java.sql.DriverManager;

/**
 * Esempio 03 - mostra il valore di default di login timeout
 */
public class Esempio03 {
   static public void main(String[] args) {
      int timeout = DriverManager.getLoginTimeout();
      System.out.println("Default Login Timeout " + timeout);
   }
}
```

`getConnection()`

Restituisce l'oggetto `Connection`. Il driver manager scorre l'elenco dei drivers caricati nella VM e su ognuno invoca il metodo `acceptsURL()` per vedere se il driver comprende il protocollo di connessione richiesto dall'applicazione. Se trova il driver adatto viene invocato il suo metodo `connect()` per ottenere la connessione.

Esistono tre forme di `getConnection()`, da utilizzare a seconda di come si vogliono passare le proprietà della connessione.

Il primo esempio:

```
DriverManager.getConnection("jdbc:informix-sqli://www.mokabyte.com:1525
/mokadb:INFORMIXSERVER=ol_mokabyte;user=mokauser;password=mokaifx");
```

Il secondo esempio:

```
DriverManager.getConnection("jdbc:informix-sqli://www.mokabyte.com:1525
/mokadb:INFORMIXSERVER=ol_mokabyte", "mokauser", "mokaifx");
```

Il terzo esempio:

```
    Properties props = new Properties();
props.put("user", "mokauser");
props.put("user", "mokaifx");

DriverManager.getConnection("jdbc:informix-sqli://www.mokabyte.com:1525
/mokadb:INFORMIXSERVER=ol_mokabyte", props);
```

Classe `Connection`

`createStatement()`

Crea l'oggetto connessione utilizzato dall'applicazione per l'esecuzione dei comandi SQL. Esiste in due forme, per il supporto della nuova funzionalità di scrolling result set.

`getTransactionIsolation()`

`setTransactionIsolation()`

Si utilizzano per impostare il livello di isolamento delle transazioni per la sessione corrente. I possibili livelli di isolamento della transazione sono riportati di seguito (in ordine di isolamento crescente, tra parentesi è riportata la relativa costante da utilizzare in `setTransactionIsolation()`).

- uncommitted read (`TRANSACTION_READ_UNCOMMITTED`). Questo livello di isolamento permette di leggere anche i dati in transazione, quindi dati che stanno quindi per essere modificati e non sono "integri".

- committed read (`TRANSACTION_READ_COMMITTED`). Lettura di soli dati non in transazione. Se un'altra sessione sta lavorando sui dati ai quali cerchiamo di accedere, la lettura viene bloccata.

- repeatable read (`TRANSACTION_REPEATABLE_READ`). Questo livello di isolamento vi garantisce non solo che i dati che leggete durante la vostra transazione siano integri ma, se all'interno della stessa transazione leggete più volte gli stessi dati (ad esempio rifate la stessa query) riotterrete gli stessi risultati. Il livello di isolamento precedente non garantisce questa possibilità.

- serializable (`TRANSACTION_SERIALIZABLE`). È il livello di isolamento più alto per una transazione utente. Oltre a ovviare ai problemi precedenti, conosciuti come

dirty read e *non-repeteable read*, permette di affrontare anche il problema dei *phantom read*, che si verifica quando un'altra transazione inserisce ad esempio una nuova riga che soddisfa a una clausola di `WHERE` di una lettura all'interno della vostra transazione. In tal caso i valori precedentemente letti non sarebbero stati modificati ma vi capiterà di vedere dati nuovi (*phantom*) che prima non comparivano nel record set.

Perché non impostare fin dall'inizio, o prima di qualsiasi transazione, il massimo livello in modo da evitare in anticipo ogni problema? La risposta è semplice: l'aumento del livello di isolamento di una o più sessioni utenti limita il grado di parallelismo del database, non riuscendo il server a eseguire le transazioni in modo concorrente. Ne risulta quindi un abbassamento delle performance. Il più delle volte è il sufficiente il livello di isolamento di default, che può essere reperito con il metodo `DatabaseMetadata.getDefaultTransactionIsolation()`. Occorre conoscere d'altra parte i diversi gradi di isolamento e quali specifiche violazioni evitano nel caso dobbiate scrivere programmi che effettuano modifiche critiche sui dati.

`commit()`

`rollback()`

`getAutoCommit()`

`setAutoCommit()`

Questi quattro metodi si utilizzano insieme per avviare una transazione in modo programmatico, senza cioè inviare al database le istruzioni SQL classiche `BEGIN WORK`, `COMMIT WORK`, `ROLLBACK`.

Normalmente tutte i comandi SQL eseguiti da un'applicazione sono svolti in modalità autocommit. Ciò vuol dire che per ogni singolo comando SQL viene aperta e chiusa una transazione in modo trasparente al client. Nel caso invece occorra rendere atomica una serie di operazioni SQL che hanno senso se eseguite insieme, disabilitare la modalità autocommit con l'istruzione

```
setAutoCommit(false); // disabilita autocommit e avvia transazione
```

Questa istruzione automaticamente avvia una transazione — non occorre insomma un'ulteriore istruzione del tipo `stmt.execute( "BEGIN WORK" )` — che sarà terminata con una delle due istruzioni `commit()` o `rollback()`.

Si vedranno in seguito esempi di utilizzo delle transazioni e dei metodi mostrati. Potete ovviamente utilizzare questi metodi solo se le transazioni per il database che state utilizzando sono supportate: per saperlo a priori utilizzate il metodo `DatabaseMetadata.supportsTransactions()`.

```
prepareStatement()
```

```
prepareCall()
```

Si utilizzano questi due metodi rispettivamente per creare uno statement preparato e una chiamata a stored procedure.

Se si prevede di utilizzare spesso la stessa istruzione SQL, conviene "prepararla", ovvero spedirla al parser del DB server in modo che possa precompilarla per poi rieseguirla con i soli parametri variabili. Ad esempio, query come la seguente

```
SELECT * FROM articoli WHERE nome = 'Nicola' AND cognome = 'Venditti'

SELECT * FROM articoli WHERE nome = 'Giovanni' AND cognome = 'Puliti'
```

possono essere preparate con la sintassi seguente

```
SELECT * FROM articoli WHERE nome = ? AND cognome = ?
```

In codice Java diventa

```
ResultSet rs = null;
PreparedStatement pstmt
= conn.prepareStatement("SELECT * FROM articoli WHERE nome = ? AND cognome = ?");

// recupera articoli di Nicola Venditti
pstmt.setString(1,'Nicola');
pstmt.setString(1,'Venditti');
rs = pstmt.executeQuery();
...

// recupera articoli di Giovanni Puliti
pstmt.setString(1,'Giovanni');
pstmt.setString(1,'Puliti');
rs = pstmt.executeQuery();
...
```

In modo simile si esegue la chiamata a una stored procedure nel database.

Classe `Statement`

Lo Statement è lo strumento che le applicazioni utilizzano per spedire comandi SQL al database. È un concetto generale valido non solo per Java.

```
execute()
```

`executeQuery()`

`executeUpdate()`

Questi, e in particolare gli ultimi due, sono i metodi da utilizzare per ogni operazione possibile sul database.

Il primo è un metodo generico, utilizzabile per istruzioni SQL tipo DDL e DML; il secondo è specializzato per le query o in generale per operazioni che ritornano un result set; `executeUpdate()`, si utilizza per tutte le operazioni che non ritornino un result set ma al più un update count (come nel caso di cancellazioni, inserimenti e aggiornamenti).

Il primo metodo è interessante anche perché permette di eseguire in una sola volta istruzioni multiple separate dal punto e virgola come

```
SELECT * FROM ARTICOLI;
DELETE FROM articoli WHERE cognome = 'Venditti';
UPDATE articoli SET nome = 'Giovanni' WHERE nome = 'Giovannino' AND cognome = 'Puliti';
SELECT count(*) FROM ARTICOLI WHERE cognome = 'Puliti';
```

Per recuperare i vari risultati si utilizzano i metodi `getMoreResults()`, `getUpdateCount()`, `getResultSet()`. Il metodo `getMoreResults()` restituisce `true` se esistono ancora risultati da leggere; nel caso ve ne siano si chiama `getUpdateCount()`. Se questa chiamata vi restituisce `-1` allora vuol dire che il risultato corrente è un result set a cui potete accedere con `getResultSet()`. Grazie a `execute()` si potrebbe implemetare un editor SQL in Java capace di eseguire blocchi di istruzioni scritti dall'utente e di visualizzarne il contenuto in modo formattato.

Per istruzioni SQL del tipo `SELECT * FROM ARTICOLI INTO TEMP temp1` è preferibile utilizzare `executeUpdate()` invece di `executeQuery()` perché la clausola `INTO TEMP` crea una tabella temporanea, quindi altera, anche se in modo non esplicito, lo schema del database, cosa che una normale Select non fa. Il valore ritornato è il numero delle righe inserite nella tabella temporanea: è appunto un update count.

Classe `ResultSet`

Il result set è un oggetto dinamicamente generato dallo statement per la manipolazione dei dati restituiti da query.

`getXXX()`

Questa famiglia di metodi, a cui appartengono ad esempio `getString()`, `getInt()`

ecc. si utilizzano per recuperare i dati presenti nel result set. Esiste un metodo `getXXX()` per ogni tipo di dato SQL.

Per la maggior parte dei campi è possibile utilizzare `getString()`, per farsi ritornare in formato di stringa il valore del risultato.

`next()`

Nelle versioni di JDBC precedenti alla 2 è possibile scandire il result set restituito dalla query solo in avanti. Per la scansione esiste dunque il solo metodo `next()`, che sposta il cursore che il database ha creato in avanti di una riga.

Si utilizzano tipicamente istruzioni come le seguenti per leggere i dati restituiti da un result set:

```
...
String sql = "SELECT * FROM ARTICOLISTI";
ResultSet rs = stmt.executeQuery(sql);

String nome = null;
String cognome = null;
int eta = 0;
while(rs.next()) {
   nome = rs.getString("nome");
   cognome = rs.getString("cognome");
   eta = rs.getInt("eta");
   ...
}
```

`wasNull()`

Supponiamo che per l'articolista "Mario Rossi" non sia pervenuta l'informazione sulla sua età. L'amministratore del DB decide, logicamente, di inserire un valore `null` nel campo della relativa riga della tabella `ARTICOLISTI`. Il valore `null` in un database non vuol dire `0` o stringa nulla ma qualcosa come "campo non valorizzato" o "valore non applicabile".

Cosa succede quando un'applicazione contenente del codice come quello appena mostrato legge l'età di "Mario Rossi"? In questi casi occorre chiamare `wasNull()`, subito dopo la lettura della colonna, in questo caso `getInt("eta")`, per capire se contiene un `null`. In caso negativo prendo il valore restituito dalla lettura.

La classe `ResultSetMetaData`

Questa classe permette di accedere a metainformazioni relative al result set. Si supponga ad esempio di eseguire la semplice query:

```
SELECT * FROM ARTICOLI;
```

Quante colonne contiene il result set restituito e quindi la tabella `ARTICOLI`? Di che tipo sono le diverse colonne? Qual è il nome delle colonne? Qual è il nome dello schema a cui appartiene la tabella `ARTICOLI`? Qual è la precisione di un eventuale campo numerico nel result set?

Si dia un'occhiata ai non molti metodi di questa classe per vedere se qualcuno di essi può essere utile nella lettura del result set.

Un'osservazione importante: non necessariamente le colonne restituite da una query coincidono o sono un sottoinsieme delle colonne della tabella che si interroga. Ecco un paio di esempi:

```
SELECT avg(eta) AS eta_media_articolisti FROM articolisti;
SELECT count(*) AS numero_articolisti FROM articolisti;
```

Classe `DatabaseMetaData`

Questa classe ha curiosamente due primati: quello della classe con più metodi e quello della classe con il metodo avente il nome più lungo di tutta la libreria Java: si può controllare il sorgente `DatabaseMetaData.java` per rendersene conto.

Il questa classe si trovano tutti metodi del tipo `supportsQualcosa` e simili: utilizzateli per ricavare tanto delle metainformazioni relative allo schema del database quanto informazioni preliminari sulle potenzialità del database stesso in termini di supporto delle caratteristiche standard.

JDBC 2.1 core API

Le versioni 2.x del JDBC introducono alcune fondamentali aggiunte che si integrano perfettamente nell'impianto originario. Il fatto che nulla del design originario sia cambiato indica quanto siano state oculate le scelte progettuali alla base di JDBC: non per tutte le librerie Java è stato così.

Le aggiunte introdotte in JDBC 2 possono essere raggruppate sotto due categorie:

nuove funzionalità:

- result sets scrollabili;
- operazioni DML programmatiche;
- batch updates;

- aggiunte minori per l'aumento delle performance, per il supporto delle time zones, ecc.

supporto per i nuovi tipi di dati:

- supporto per i nuovi tipi astratti SQL3;
- supporto per l'archiviazione diretta degli oggetti nel database.

Si analizzeranno ora più nel dettaglio le nuove funzionalità.

Nuove funzionalità di JDBC

Result set scrollabili

Chi è abituato a lavorare con Access, per fare un esempio, trova naturale eseguire una query, recuperare il record set (secondo la terminologia di Microsoft) e scandirlo in avanti o all'indietro, posizionare il cursore dove si vuole o avere il count degli elementi. Questo, almeno con JDBC 1.1, non è possibile: una volta recuperato il result set lo si può scandire solo in una direzione e non è possibile avere il count delle righe restituite. Perché con Access è possibile fare una cosa in apparenza così semplice che con JDBC sembra impossibile?

La limitazione sta in parte in JDBC e in parte nel meccanismo con cui il database restituisce i dati ai client. Come accennato altrove il result set, dal punto di vista del database è un cursore posizionato all'inizio della tabella fittizia che contiene i risultati che soddisfano alla query. Una volta spostato in avanti il cursore non è possibile ritornare indietro normalmente: normalmente vuol appunto dire che il tipo di cursore allocato per l'operazione di lettura dei risultati è unidirezionale e questo perché è il meno impegnativo dal punto di vista delle risorse.

Quasi tutti i database, compresi Oracle e Informix utilizzano questo metodo. In alcuni casi, per venire incontro ai programmatori e fare in modo che possano utilizzare comunque questa possibilità, i produttori di database hanno fornito dei drivers ODBC con un'opzione impostabile dall'amministratore della sorgente di dati ODBC per l'uso dei record set scrollabili.

Con JDBC 2 il programmatore Java ha la possibilità di scegliere programmaticamente l'uso di questa opzione, specificandolo nel normale processo di creazione dello statement. Ad esempio:

```
...
Connection conn
    = DriverManager.getConnection("jdbc:informix-sqli://www.mokabyte.it:
    1526/mokadb:INFORMIXSERVER=ol_mokabyte", "informix", "mokapass");
```

```
Statement stmt
= conn.createStatement(ResultSet.SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_READ_ONLY);

...
```

In questo caso creo un cursore insensibile ai cambiamenti apportati da altri sullo stesso result set, e di sola lettura.

Il supporto completo per lo scrollable result set è così articolato:

- L'oggetto `ResultSet` è stato dotato di metodi opportuni per il posizionamento casuale del cursore oltre che di attributi da utilizzare come costanti per la definizione per tipo di result set che si vuole.

- All'oggetto `Connection` sono stati aggiunti degli overloading per il metodo `createStatement()` per permettere al programmatore di poter definire il tipo di cursore desiderato.

- All'interfaccia `DatabaseMetaData` sono stati aggiunti metodi perché l'applicazione possa accertarsi a priori se il database supporta questa funzionalità.

Cancellazione, inserimento e updates programmatici

La maniera tradizionale di aggiornare una riga di una tabella è quella di eseguire un comando SQL di `UPDATE`. Ad esempio la seguente istruzione:

```
...
stmt.executeUpdate("UPDATE prodotti SET prezzo
                   = 2500 WHERE nome_prodotto = 'BIRRA'");
...
```

aggiorna il prezzo di un prodotto in catalogo; allo stesso modo si inserisce un nuovo prodotto o se ne cancella uno.

JDBC 2 introduce un modo alternativo di eseguire le istruzione di tipo DML come quelle viste. Per le operazioni di aggiornamento dei dati, JDBC mette ora a disposizione le istruzioni `updateXXX()`, che agiscono sui campi della riga corrente. I passi che il programma segue per aggiornare i dati è il seguente:

1. Connessione e creazione del result set (*updatables*).

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");   // JDBC / ODBC
Connection conn =  DriverManager.getConnection("jdbc:odbc:mokadb");
```

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);
   ResultSet srs = stmt.executeQuery("SELECT * FROM articoli");
```

2. Posizionamento sulla riga che si vuole aggiornare e aggiornamento dei campi che interessano.

```
srs.first();
// aggiorna il titolo del primo articolo di MokaByte
srs.updateString("TITOLO", "Primo articolo di MokaByte!");
```

3. Aggiornamento della riga.

```
srs.updateRow();   // Conferma l'update
```

4. Chiude la connessione.

```
srs.close();
stmt.close();
conn.close();
```

Nel caso si volessero annullare tutte le operazioni di aggiornamento fatte, prima di `updateRow()`, è sufficiente chiamare `ResultSet.cancelRowUpdates()`, e saranno ripristinati i valori dei campi al valore originario.

Nel caso di inserimento di una nuova riga si procede invece come segue:

1 Connessione e creazione del result set (*updatables*).

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// via ODBC
Connection conn =  DriverManager.getConnection("jdbc:odbc:mokadb");
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
   ResultSet srs = stmt.executeQuery("SELECT * FROM articoli");
```

2. Ci si posiziona sulla “riga di inserimento” — una riga fittizia introdotta per rendere possibile l’operazione di inserimento — con il nuovo metodo `ResultSet.moveToInsertRow()`, si usano i metodi `updateXXX()` per definire i campi della nuova riga e, infine, si ritorna alla precedente riga del result set.

```
srs.moveToInsertRow();
// imposta il titolo
```

```
srs.update("TITOLO", "Primo articolo di MokaByte!");
// id autore (nella tabella autori)
srs.update("IDAUTORE", 1200);
// data articolo
srs.update("DATA", "10/10/2000");
...
// inserisce la riga nella tabella ARTICOLI
srs.insertRow();
// ritorna alla precedente riga del result set
srs.moveToCurrentRow();
```

3. Chiude la connessione

```
srs.close();
stmt.close();
conn.close();
```

L'istruzione `srs.moveToCurrentRow()`, permette di ritornare alla riga corrente del result set prima della chiamata a `moveToInsertRow()`.

Ancora più semplice dell'inserimento di una riga è la sua cancellazione. Le istruzioni sono le seguenti:

```
// ci si posizione sulla riga contenente la riga da cancellare
srs.absolute(5):
// si cancella la riga dalla tabella
srs.deleteRow();
```

Possono sorgere delle inconsistenze nel result set corrente quando si cancellano o si inseriscono delle righe. Per esempio, nel caso di cancellazione di una riga, essa non dovrebbe più comparire nel result set. Ma alcuni driver JDBC introducono in luogo della riga cancellata una riga "blank" come rimpiazzo.

In generale, per fronteggiare problemi come questi occorre interrogare i metodi `ownUpdatesAreVisible()`, `ownDeletesAreVisible()` e `ownInsertsAreVisible()` dell'interfaccia `DatabaseMetaData`, che danno informazioni sul supporto fornito dal driver JDBC per le funzionalità da utilizzare.

Nel caso specifico è comunque possibile chiudere e riaprire il result set (attraverso una nuova query) per evitare qualsiasi problema di inconsistenza dei dati.

Batch updates

Questa nuova possibilità, introdotta più per ragioni di performance che come funzionalità applicativa, permette di spedire al database una serie di updates in blocco (o in "batch") piuttosto che uno dopo l'altro: si da così al database la possibilità di introdurre delle ottimizzazioni

Anche in questo caso sono state modificate alcune classi per poter supportare questa aggiunta: le classi `Statement`, `PreparedStatement`, and `CallableStatement` hanno dei metodi in più, vale a dire `addBatch()`, `clearBatch()`, `executeBatch()`; alla classe `DataBaseMetaData` è stato aggiunto il metodo `supportsBatchUpdates()`; infine è stata introdotta una nuova eccezione, ovvero `BatchUpdateException`, lanciata nel caso di anomalie nel processo di batch update.

Ecco un esempio di codice che esegue una serie di operazioni DML in batch per introdurre tre nuovi articoli di un nuovo articolista di MokaByte.

```
try {
// apre una transazione
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
// inserisce dati anagrafici dell'articolista
stmt.addBatch("INSERT INTO AUTORI " + "VALUES(1001, 'Mario', 'Rossi',
                                             'marior@mokabyte.it')");
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('Enterprise JavaBeans Parte 1',
                                               1001, '10/06/2000')");
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('Enterprise JavaBeans Parte 2',
                                               1001, '10/07/2000')");
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('EJB e J2EE', 1001, '10/08/2000')");

// esegue batch
int [] updateCounts = stmt.executeBatch();
// chiude la transazione
conn.commit();
// default auto-commit
conn.setAutoCommit(true);

} catch(BatchUpdateException be) {
   System.err.println("---- BatchUpdateException ----");
System.err.println("SQLState: " + be.getSQLState());
System.err.println("Messaggio err.: " + be.getMessage());
   System.err.println("Codice err.: " + be.getErrorCode());
System.err.print("Updates: ");
   int [] updateCounts = be.getUpdateCounts();
   for (int i = 0; i < updateCounts.length; i++) {
      System.err.print(updateCounts[i] + " ");
   }
} catch(Exception e) {
   System.err.println("--- EXCEPTION! ---");
   System.err.println(e.getMessage());
}
```

Anche per questa funzionalità è bene accertarsi che sia effettivamente supportata dal driver attraverso il metodo `DatabaseMetaData.supportsBatchUpdates()`.

Supporto per i tipi di dati avanzati

Tipi di dati SQL3 in Java

Per nuovi tipi di dati SQL3 si intendono i nuovi tipi di dato SQL introdotti dallo standard SQL3. SQL ha attraversato una serie di successive standardizzazioni: ISO/ANSI, SQL86, SQL89, SQL92 ed ora SQL3.

Con SQL2 e SQL3 sono stati introdotti i seguenti nuovi tipi di dati non elementari:

- `CLOB` (Character Large OBject). È un cosiddetto *smart blob space* perché, a differenza dei precedenti, forme di large object permettono un accesso casuale al dato stesso. Si utilizza per l'immagazzinamento lunghe stringhe di caratteri (diciamo qualche kylobyte) come ad esempio descrizioni estese, ecc.

- `BLOB` (Binary Large OBject). Si utilizza per immagazzinare dati binari molto lunghi come immagini, documenti in formato binario, ecc.

- `ARRAY`. Permette di utilizzare un array come valore di una colonna.

- `ROW TYPE`. Definisce il tipo di dato riga. È possibile ad esempio che una colonna di database sia strutturata, contenendo a sua volta una riga all'interno e violando così la prima forma normale del modello relazionale (come nel caso di un array).

- `REF`. Si tratta del tipo di dato "riferimento"; serve cioè per definire riferimenti a dati.

Per ognuno di questi nuovi tipi di dato SQL, Java ha un oggetto o un'interfaccia che ne permette l'utilizzo nei programmi. Precisamente:

- un oggetto `Clob` corrisponde a un `CLOB SQL`;

- un oggetto `Blob` corrisponde a un `BLOB SQL`;

- un oggetto `Array` corrisponde a un `ARRAY SQL`;

- un oggetto `Struct` corrisponde a un `ROW TYPE`;

- un oggetto `Ref` corrisponde a un `REF SQL`.

Esiste ancora un altro tipo di dato SQL non citato in precedenza: il `DISTINCT TYPE`. In molti database è presente già da molto. Non è indispensabile e normalmente lo si utilizza solo per affinare logicamente il modello della propria base dati. Un distinct type è, grosso modo, una ridefinizione di un tipo di dato primitivo esistente in un range di valori diverso dal default. Ad esempio:

```
CREATE DISTINCT TYPE CODART_T AS CHAR(8);
```

crea il nuovo tipo di dati `COD_ARTICOLO` che è un `CHAR` di esattamente 8 caratteri. Non esistendo il tipo `CHAR(8)` è indubbiamente una comodità poterlo definire prima di disegnare il modello del database. In tal modo infatti posso scrivere

```
— tabella articoli di MokaByte!
CREATE TABLE articoli
(

...,
CODICE CODART_T, — il codice articolo è di tipo CODART_T alias CHAR(8)
...
);
```

Se non esistessero i distinct type si dovrebbe aggiungere un costraint a ogni tabella che contiene il codice dell'articolo oppure definire un dominio:

```
CREATE DOMAIN CODART CHAR(8);
```

Dal punto di vista applicativo il tipo distinct type non ha corrispondenza e, per recuperare il valore di una colonna contenente un distinct type, è sufficiente far riferimento al tipo di dato originario su cui esso è definito. Nel nostro esempio quindi, per avere un banale report codice–titolo, si scriverà

```
...
ResultSet rs = stmt.executeQuery("SELECT CODICE, TITOLO FROM ARTICOLI");
System.out.println("CODICE | TITOLO");
while(rs.next()) {
   System.out.println(rs.getString("CODICE") + "\t" + rs.getString("TITOLO"));
}
...
```

Ecco ora come si utilizzano gli oggetti introdotti per manipolare i nuovi tipi SQL3.

Clobs, Blobs, Arrays

Oltre ad aver introdotto le interfacce `Clob`, `Blob` e `Array`, JDBC 2 ha ovviamente esteso la classe `ResultSet` aggiungendovi i metodi corrispondenti `getClob`, `setClob` ecc., tramite i quali effettivamente manipolare i dati.

Da notare che `Clob` e `Blob` sono appunto delle interfacce, quindi non è possibile istanziare direttamente un oggetto di uno di questi tipi. Sarebbe davvero comodo se potessimo istanziare un `Blob` passandogli ad esempio il nome di un file contenente un'immagine da immagazzinare nel database

```
Blob b = Blob("/usr/home/nico/immagini/mokabyte.jpg");
```

e passare poi l'oggetto a un metodo `setBlob()` che lo inserirà nella colonna.

Esistono metodi proprietari per inserire clob e blob in una colonna. Con Java si utilizzano i metodi `ResultSet.setBinaryStream()` / `ResultSet.setObject()` per inserire blob e `ResultSet.setAsciiStream()` per inserire clob.

Ecco le definizione delle interfacce `Blob` e `Clob`.

```
package java.sql;
public interface Blob {
   long length() throws SQLException;
   InputStream getBinaryStream() throws SQLException;
   byte[] getBytes(long pos, int length) throws SQLException;
   long position(byte [] pattern, long start) throws SQLException;
   long position(Blob pattern, long start) throws SQLException;
}


package java.sql;
public interface java.sql.Clob {
   long length() throws SQLException;
   InputStream getAsciiStream() throws SQLException;
   Reader getCharacterStream() throws SQLException;
   String getSubString(long pos, int length) throws SQLException;
   long position(String searchstr, long start) throws SQLException;
   long position(Clob searchstr, long start) throws SQLException;
}
```

Si supponga che nel database di MokaByte la tabella articoli abbia una colonna `documento` contenente l'articolo in formato MS Word. Per accedervi si utilizzano le istruzioni che seguono:

```
...
// query
String sql = "SELECT idarticolo, titolo, documento FROM articoli, "
             + " WHERE idarticolista = 1000 AND date = '12/02/1999'";
// eseguo query
ResultSet rs = stmt.executeQuery(sql);
// so di recuperare un solo articolo!
rs.first();
```

```
String titolo = rs.getString("titolo");
// ottengo blob
Blob data = rs.getBlob("documento");
...
```

A questo punto per recuperare il contenuto del documento `.doc` del nostro articolista e scriverlo su un file si utilizzano i metodi di `Blob`:

```
OutputStream doc
= new  BufferedOutputStream(new FileOutputStream(titolo + ".doc"));
InputStream in = blob.getBinaryStream();
byte b;
// copia dati
while ((b = in.read()) > -1) {
   doc.write();
}
// chiusura streams
doc.close();
in.close();
// log attività
System.err.println("Documento #" + rs.getString("idarticolo")
                     + " in file " + titolo + ".doc");
```

Con i Clobs si opera in modo del tutto simile.

L'interfaccia `Array` viene utilizzata per manipolare il tipo di dati SQL `ARRAY`. Per creare un tipo di dato `ARRAY` nel database si utilizza la seguente sintassi SQL:

```
CREATE TYPE INTERESSI_T AS ARRAY(10) OF VARCHAR(40);
```

Lo si utilizza poi quando si crea una tabella; ad esempio:

```
CREATE TABLE ARTICOLISTI
(
   idarticolista   SERIAL,
   nome            VARCHAR(30);
   ...
   interessi       INTERESSI_T, — array interessi (es.: 'musica', 'palestra', ...)
   ...
);
```

Si può aggiungere la colonna alla tabella, se già esiste:

```
ALTER TABLE ADD interessi INTERESSI_T;
```

È stata dunque aggiunta alla tabella `articolisti` del nostro database degli articoli di MokaByte una colonna `interessi`, di tipo `ARRAY`, che comprende un elenco di interessi che danno un po' di informazioni al profilo utente dell'articolista, in particolare le sue preferenze per il tempo libero. Dunque per recuperare queste informazione dovranno essere eseguite le seguenti istruzioni:

```
...
String sql = "SELECT interessi FROM articolisti "
            + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet. rs = stmt.executeQuery(sql);
Array arr = (String[])rs.getArray(1);
String interessi = arr.getArray();

System.out.println("Preferenze dell'articolista " + nome + " " + cognome + ": ");
for(int = 0; i < interessi.length(); ++i) {
   System.out.println(" " + i + ")" + interessi[i]);
}
...
```

Per l'inserimento invece occorre utilizzare la sintassi SQL per gli `ARRAY`.

Tipo REF

Come detto il tipo `REF` è un tipo di dato utilizzato per referenziare il dato vero e proprio. Il suo utilizzo è necessario per evitare inutili duplicazioni di dati. Ecco un esempio di utilizzo del tipo `REF` all'interno del database di MokaByte.

MokaByte organizza periodicamente dei corsi itineranti per la diffusione e l'insegnamento del linguaggio Java. Per questi corsi vengono di volta in volta "arruolati" alcuni articolisti come docenti. Si vuole tenere traccia nel database di MokaByte di tutta questa attività e per farlo introduciamo un paio di tabelle: `JAVA_TOUR` e `CORSO_TOUR`: la seconda tabella server per raccogliere le informazioni sui corsi proposti la prima invece, una tabella molti a molti associa un corso a un'articolista (che diventa così docente per quel corso) in una certa data e per un certo tour.

```
— Tabella dei corsi su Java
CREATE TABLE corsi OF CORSO (OID REF(CORSO) VALUES ARE SYSTEM GENERATED);

— Tabella dei tours
CREATE TABLE java_tours
(
   idtour INTEGER,          — un id diverso per ogni tour (non PK!)
   descr VARCHAR(200),      — descrizione di questo tour
   iddocente INTEGER,       — id articolista che si occupa di un corso nel tour
   corso REF(corsomoka),    — corso sostenuto in questo tour
```

```
   data DATE,              — data del corso
   sponsors SPONSORS_T     — array degli sponsors
 FOREIGN KEY(iddocente) REFERENCES articolisti(idarticolista)
   ON DELETE SET NULL ON UPDATE CASCADE
);
```

La tabella corsi è creata a partire dal tipo `corso`

```
CREATE TYPE corsomoka
(
   idcorso     INTEGER,          — id
   titolo      VARCHAR(20),      — titolo del corso
   descr       VARCHAR(200)      — descrizione corso
   costo       DECIMAL(10, 3)    — costo del corso
   argomenti   ARGOMENTI_T       — un array per gli argomenti!
);
```

A questo punto vediamo come utilizzare l'interfaccia `Ref` per recuperare tutti i corsi relativi al primo tour di MokaByte.

```
Statement pstmt
= conn.prepareStatement("SELECT * FROM corsi WHERE oid=?");
Statement stmt = conn.createStatement();
String sql
= "SELECT corso FROM java_tours WHERE nome = 'Primo Java Tour di MokaByte'";

ResultSet rs1 = stmt.executeQuery(sql);
ResultSet rs2 = null;
Ref curr =null;
while(rs1.next()) {
   curr = rs1.getRef(1);
   pstmt.setRef(1, curr);
   rs2 = pstmt.execute();
   System.out.println("");
   System.out.println("---------------------------------");
   System.out.print(rs2.getString(2));   // titolo corso
   System.out.print("---------------------------------");
   System.out.println("");
   System.out.println("Descr:" + rs2.getString(3));
   System.out.println("Costo:" + rs2.getString(4));
   System.out.print("Argomenti: "
   Array arr = rs2.getArray(5);
   String[] argomenti = (String[])arr.getArray();
   for(int i=0; i<argomenti.length;+i){
      System.out.print (argomenti[i]);
      if(i <argomenti.length) System.out.print(",");
   }
```

```
   System.out.println("");
   rs2.close();
}
```

Nell'esempio era sufficiente mettere in join le due tabelle per arrivare allo stesso risultato

```
"SELECT c.corsi FROM corsi c, java_tours j
WHERE j.nome = 'Primo Java Tour di MokaByte' AND j.oid = c.corso"
```

Struct

L'interfaccia `Struct` serve per mappare i tipi di dato SQL3 `ROW TYPE`. L'esempio più classico è quello di associare all'entità indirizzo di un database anagrafico row types del tipo:

```
CREATE ROW TYPE INDIRIZZO_T   — (si utilizza anche CREATE TYPE...)
(
   citta           VARCHAR(40),
   via             VARCHAR(40),
   numero          VARCHAR(8),    — numero civico alfanumerico; p.e.: 25/A
   telefono        CHAR(10));
```

Ed eccone un esempio di utilizzo all'interno di una tabella.

```
CREATE TABLE articolisti
(
   idarticolista   SERIAL,      — Informix SERIAL in integer autoincrementantesi
   nome            VARCHAR(20),
  ...
   indirizzo       INDIRIZZO_T  — riga all'interno di una colonna!
);
```

Il row type rappresenta appunto una riga che può essere annidata, se si utilizza come nell'esempio precedente, all'interno di un'altra riga che rappresenta un'informazione strutturata che logicamente non andrebbe separata dall'entità alla quale si riferisce.

Per trattare dati così strutturati Java mette a disposizione l'interfaccia `Struct`. Ecco un esempio di utilizzo.

```
...
String sql = "SELECT indirizzo FROM articolisti "
             + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet rs = stmt.executeQuery(sql);
if(rs.netx()) {
   Struct indirizzo = rs.getStruct(1);
```

```
   String[] attr = (String[])indirizzo.getAttributes();
   System.out.println("Indirizzo di Nicola Venditti: ");
   System.out.println("\t Citta: " + attr[1]);
   System.out.println("\t Via: " + attr[2]);
   System.out.println("\t Telefono: " attr[4]);
}
...
```

Il metodo `Struct.getAttributes()` restituisce tutti i campi in formato di array di oggetti.

Serializzazione degli oggetti Java nel database

L'introduzione del tipo `Struct` permette di utilizzare i tipi di dato complessi: siamo quasi un passo da un oggetto vero e proprio.

Perché ci sia una vera corrispondenza logica tra tipo strutturato SQL (row type) e tipo Java occorrerebbe che l'oggetto restituito da `ResultSet.getXXX()` fosse direttamente un oggetto Java di classe adeguata; ad esempio, per continuare con l'esempio sopra riportato, occorrerebbe che il result set restituisse un oggetto di classe `Indirizzo` con attributi simili a quelli di `INDIRIZZO_T`.

Questa possibilità esiste ed è indicata dalla documentazione del JDBC come *custom mapping*. È possibile cioè indicare al driver JDBC quale classe si vuole utilizzare e si vuole che sia restituita in corrispondenza di un dato strutturato. Fatto ciò sono sufficienti una chiamata al metodo `ResultSet.getObject()` per ottenere un riferimento all'oggetto della classe di mappatura e una conversione di cast per avere l'oggetto finale. Traducendo quanto appena detto in un esempio:

```
...
Connection conn = DriverManager.getConnection(user, passwd);

// ottiene mappa tipi
java.util.Map map = conn.getTypeMap();

// aggiunge il tipo personalizzato INDIRIZZO_T
map.put("'mokabyte'.INDIRIZZO_T", Class.forName("Address");
...
String sql = "SELECT indirizzo FROM articolisti "
            + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet rs = stmt.executeQuery(sql);
   rs.next();

// recupera il row type come oggetto Indirizzo e non come
// generica Struct
Indirizzo indirizzo = (Indirizzo)rs.getObject("indirizzo");
```

Il codice mostrato apre la connessione, ottiene la mappa di corrispondenza dei tipi (tipo SQL <—> tipo Java), aggiunge un'entry per il tipo `INDIRIZZO_T` e infine utilizza `getObject()` e un semplice cast per ricavare l'oggetto `indirizzo` di tipo `Indirizzo`.

La classe Java `Indirizzo` non può essere scritta liberamente ma deve sottostare ad alcune regole. In particolare, deve implementare l'interfaccia `java.sql.SQLData` di JDBC 2. Per la verità ogni DBMS ha uno strumento che, a partire dal tipo row type nel database, genera il codice per una classe di mappatura. Nel nostro caso la classe potrebbe essere scritta così

```
public class Indirizzo implements SQLData {
   public String citta;
   public String via;
   public int numero;
   public int telefono;
   private String sql_type;
   public String getSQLTypeName() {
      return sql_type;
   }

   public void readSQL(SQLInput stream, String type) throws SQLException {
      sql_type = type;
      citta = stream.readString();
      via = stream.readString();
      numero = stream.readInt();
      telefono = stream.readString();
   }

   public void writeSQL(SQLOutput stream) throws SQLException {
      stream.writeString(citta);
      stream.writeString(via);
      stream.writeInt(numero);
      stream.writeString(telefono);
   }
}
```

La mappa dei tipi così definita è a livello di connessione, quindi tutti i metodi utilizzeranno questa per le conversioni di tipo; tuttavia esiste anche la possibilità di definire mappe alternative da passare ad alcuni metodi che supportano questa possibilità.

Estensioni Standard di JDBC 2 (Optional Package)

Durante lo sviluppo di JDBC 2.0 ci si rese conto che, per varie ragioni, era opportuno dividere l'intera API in due parti: una parte fu chiamata JDBC Core API e l'altra JDBC Extension API (ora Optional Package).

L'intenzione dei progettisti di JDBC era di separare quella parte dell'API che formava il nucleo o *core*, da quelle parti che venivano aggiunte per soddisfare le esigenze di integrazione con la sempre crescente J2EE da un lato e la richiesta di nuove funzionalità da parte della comunità degli sviluppatori dall'altro. Per evitare di dover rilasciare una nuova versione di JDBC ogni volta che si aggiungono nuove funzionalità o nuovi moduli, si è scelto di seguire questa strada: di separare ciò che è ormai consolidato (parliamo per lo più di un'API per tutti i database di tipo OLTP) da ciò che costituisce solo un'aggiunta o un arricchimento del core. Cosa succede nel caso ad esempio si voglia aggiungere (e lo si farà presto) il supporto per il data-warehousing in JDBC? Semplice: verrà creato un package aggiuntivo (una Standard Extension quindi) diciamo `javax.olap.*`, che conterrà tutta le classi utili per accedere e trattare con i data-warehouse.

Estensioni standard di JDBC 2.1

Attualmente (JDBC 2.1) le estensioni standard sono tre e sono state introdotte per lo più per l'integrazione di JDBC nella piattaforma Java 2 Enterprise Edition.

In quest'ultima, ogni sorgente di dati come il database (ma può anche essere una repository molto più banale come un file system ad esempio) viene visto come una sorgente di dati o secondo la terminologia JNDI come un Data Source.

Ecco quindi che sorge la necessità di integrare JDBC nel meccanismo dei nomi di JNDI, per fare in modo che il database sia raggiungibile con un nome e quindi le applicazioni Java si riferiscano ad esso con questo nome, guadagnando così un grado di astrazione che permette alle applicazioni stesse di essere del tutto portabili tra due piattaforme J2EE compliant.

L'estensione di JDBC per il supporto delle transazioni distribuite nasce anche dall'esigenza di integrazione con un altro componente di J2EE e precisamente l'interfaccia Java Transaction Api, che permette alle applicazioni di avviare una transazione su più database che supportano questa funzionalità (occorre che supportino nativamente il protocollo XA per le transazioni distribuite, oltre a fornirne un'implementazione attraverso il loro driver JDBC).

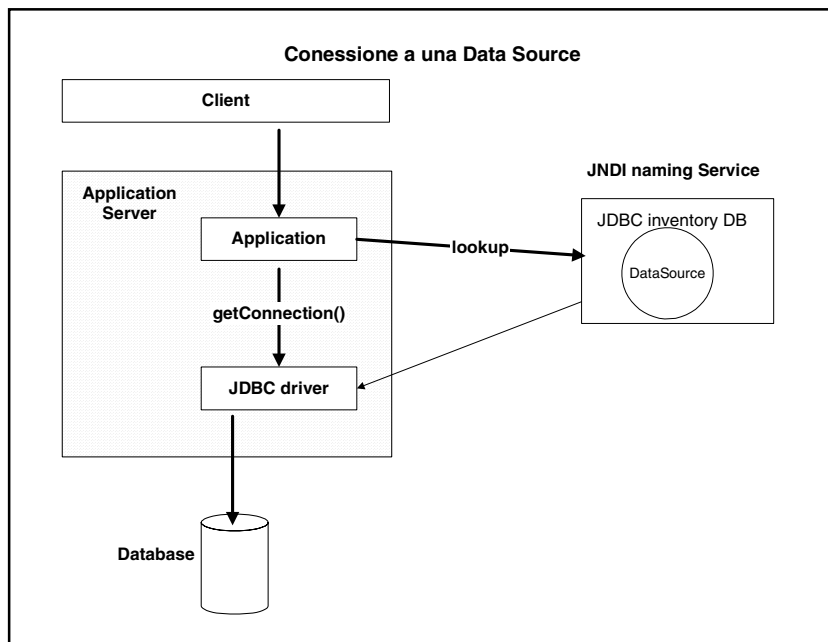
Le altre due estensioni standard di JDBC sono state introdotte per lo più per ragioni di performance e flessibilità.

Il meccanismo del Connection Pooling, che è stato sicuramente molto apprezzato da tutti gli sviluppatori, dà la possibilità di utilizzare un componente intermedio per l'accesso al database che si occupa di gestire le connessioni (in tal senso è un Connection Manager) pre-allocandone un certo numero e distribuendole (pooling) su richiesta dei client.

L'altra estensione, introdotta da JavaSoft è quella dei RowSets che dà la possibilità di avere degli oggetti che somigliano a dei result sets ma possono ad esempio essere disconnessi dal database e che in più permettono una maggiore integrazione dei dati nel modello a componenti basato su JavaBeans.

Ecco di seguito le interfacce introdotte per supportare le funzionalità appena discusse, da notare come siano tutte poste in packages del tipo `javax.sql*` cioè appunto packages di estensione della VM.

```
javax.sql.DataSource

javax.sql.XAConnection
javax.sql.XADataSource

javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSource
javax.sql.PooledConnection

javax.sql.RowSet
javax.sql.RowSetEvent
javax.sql.RowSetInternal
javax.sql.RowSetListener
javax.sql.RowSetMetaData
javax.sql.RowSetReader
javax.sql.RowSetWriter
```

Figura 7.3 – *Accesso a una sorgente di dati*

Si vedrà ora un po' più in dettaglio quanto introdotto, a partire dal supporto per JNDI.

JNDI Data Sources

Visto che l'argomento è già stato introdotto in questa sezione si vedrà ora come si utilizza JNDI per accedere al database.

JNDI è una delle enterprise API fornite da J2EE che permette alle applicazioni di accedere a servizi standard dalla piattaforma. JNDI permette di accedere al servizio di nomi e di directory che permette a sua volta di raggiungere risorse di vario tipo attraverso i due diffusi meccanismi di lookup e browsing, due metodi di ricerca, il primo dei quali utilizzato quando si conosce il nome della risorsa (o almeno un pattern per il nome) il secondo invece utilizzato quando si conoscono degli attributi della risorsa.

Si torni per un attimo alla maniera tradizionale in cui una applicazione accede a una sorgente di dati.

Per prima cosa si carica il driver JDBC

```
Class.forName("com.informix.jdbc.IfxDriver");
```

poi viene creata la connessione,

```
Connection con
= DriverManager.getConnection("jdbc:informix-sqli://dbserver.mokabyte.it
:1526/mokadb:INFORMIXSERVER=ol_mokabyte;USER=mokauser;PASSWORD=mokaifx");
```

il resto è infine normale programmazione JDBC.

Questo metodo porta con sé la conseguenza evidente che occorre sapere la locazione del database ci si connette, ma se cambia per qualche ragione anche solo la porta su cui ascolta il database server bisogna rimettere mano al codice.

Certo bisogna modificare quasi sempre solo una linea di codice, ma qui si vuole evidenziare che manca quel passaggio intermedio per togliere ogni riferimento alla base di dati vera e propria.

Perché si possa raggiungere questa separazione occorre che le applicazioni possano contare su un ambiente esterno che offra dei servizi come il JNDI per mezzo del quale un nome può rappresentare una sorgente di dati. In tal modo l'applicazione potrà essere distribuita con un file di proprietà o di configurazione ad esempio che indica su quale base dati lavora e quale nome dare a quest'ultima. Se qualcosa cambia nell'URL per l'accesso al database o cambia proprio il database, basta reimpostare alcune proprietà del file: l'applicazione continuerà a utilizzare lo stesso nome per connettersi però a una diversa base dati. Si fa in modo insomma di demandare alla fase di deployment quegli aspetti preliminari e di contorno non legati alla logica applicativa. Questo lavoro per fortuna è già stato fatto e standardizzato, e si tratta appunto di Java 2 Enterprise Edition.

Ecco come ci si connette a una base di dati utilizzando il nome ad essa associato, in un'applicazione J2EE.

```
...
Context ctx = new InitialContext();
DataSource ds =(DataSource)ctx.lookup("jdbc/mokadb");
Connection conn = ds.getConnection("mokauser", "mokaifx");
...
```

Nessun driver da precaricare, nessun riferimento a parametri fisici come il dbserver, la porta del socket ecc.

Per prima cosa si crea un `Context` per accedere al servizio dei nomi, poi si ottiene un'interfaccia `DataSource` verso la base dati e, infine, si ottiene la connessione attraverso il metodo `DataSource.getConnection()`. Per funzionare, tutto questo ha bisogno di un lavoro preliminare che prepari l'ambiente in cui l'applicazione sarà installata ed eseguita.

Questa fase preliminare si riassume in tre passi:

- creazione di un oggetto `DataSource`;
- impostazione delle proprietà del `DataSource`;
- registrazione del `DataSource` presso un servizio dei nomi (per lo più JNDI in J2EE).

Ecco del codice che esegue il primo e il secondo compito, supponendo di avere un nostro driver JDBC che supporta il meccanismo dei `DataSource`:

```
it.mokabyte.MokaDataSource ds = new it.mokabyte.MokaDataSource();
ds.setServerName("ol_mokabyte");
ds.setDatabaseName("mokadb");
ds.setDescription("Database di tutti gli articoli di MokaByte");
```

Adesso la parte più importante, la registrazione della nostra fonte di dati presso il servizio dei nomi utilizzando il nome `mokadb`, che tutte le applicazioni client utilizzeranno per riferirsi al database degli articoli:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
```

Il nome completo della risorsa è `"jdbc/mokadb"` è non solo `mokadb`, perché JNDI ha una struttura gerarchica dei nomi, e quindi occorre in generale l'intero percorso per

riferirsi a una risorsa, altrimenti il naming service non riuscirà a trovare ciò che stiamo cercando.

Per concludere questa sezione ancora due parole sugli oggetti `DataSource`. Vi sono diverse possibilità o per meglio dire livelli di implementazione di un oggetto `DataSource`:

- una implementazione di base di `DataSource` che fornisce oggetti standard `Connection` non soggetti a pooling né utilizzabili in transazioni distribuite.
- una implementazione della classe `DataSource` che supporta il connection pooling producendo oggetti `Connection` riciclabili nel senso del Connection Pool Manager. L'interfaccia interessata è `ConnectionPoolDataSource`.
- una implementazione completa di `DataSource` che supporta rispetto alla precedente anche le trasazioni distribuite fornendo oggetti `Connection` utilizzabili in una transazione distribuita. L'interfaccia interessata è `XADataSource`.

Nel caso di pooled `DataSource` si userà codice simile al seguente:

per registrare il Connection Pool Manager…

```
// Crea il Pool Manager
it.mokabyte.ConnectionPoolDataSource cpds = new it.mokabyte.ConnectionPoolDataSource();

// Imposta le proprietà
cpds.setServerName("ol_mokabyte");
cpds.setDatabaseName("mokadb");
cpds.setPortNumber(1526);
cpds.setDescription("Connection pooling per il database degli articoli di MokaByte.");
// Registra presso il servizio dei nomi il Pool Manager
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/mokadbCP", cpds);
```

per registrare il DataSource…

```
it.mokabyte.PooledDataSource ds = new it.mokabyte.PooledDataSource();
ds.setDescription("Pooled Connections per il DB degli articoli di MokaByte");
ds.setDataSourceName("jdbc/pool/mokadbCP");
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
```

Nel caso invece di DataSource che supporta pooling e transazioni distribuite si userà il seguente codice:

per registrare il Distributed Transaction DataSource…

```
// istanzia oggetto che implementa XADataSource
it.mokabyte.XATransactionalDS xads = new it.mokabyte.XATransactionalDS();
// impostazioni
xads.setServerName("mokadb");
xads.setDatabaseName("mokadb");
xads.setPortNumber(1526);
xads.setDescription("Manager per transazioni distribuite");
// registrazione
Context ctx = new InitialContext();
ctx.bind("jdbc/xa/mokadbXA", xads);
```

per registrare il DataSource…

```
// istanzia oggetto che implementa DataSource
it.mokabyte.TransactionalDataSource ds
= new it.mokabyte.TransactionalDataSource();
// impostazioni
ds.setDescription("DataSource per Transazioni Distribuite");
ds.setDataSourceName("jdbc/xa/mokadbXA");
// binding della risorsa al nome logico
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
```

Gli oggetti di classe `it.mokabyte.PooledDataSource` e `it.mokabyte.TransactionalDataSource` che implementano l'interfaccia `DataSource` sono ovviamente studiati per lavorare con i corrispettivi oggetti di tipo `it.mokabyte.ConnectionPoolDataSource` e `it.mokabyte.XATransactionalDS`, che impementano a loro volta le interfacce più specializzate `ConnectionPoolDataSource` e `XADataSource`.

Concludendo, si può dire che, dove possibile, un'applicazione Java dovrebbe utilizzare un `DataSource` invece del tradizionale metodo basato su `DriverManager` per accedere ai dati, per guadagnare in portabilità ed eleganza definendo gli aspetti più prettamente sistemistici di essa durante la fase di deployment.

Connection Pooling

Nella precedente sezione si è visto come registrare un pooled DataSource e come utilizzarlo. Si è visto quindi che dal punto di vista dell'applicazione il pooling delle connessioni è del tutto trasparente.

Un Connection Pool è una cache di connessioni a database gestita da un componente, detto Connection Manager, solitamente fornito dal JDBC Driver Vendor insieme al driver

stesso, che si occupa di recuperare da tale cache una connessione quando un client ne faccia richiesta e deallocarla quando un client non ne ha più bisogno.

Dato che il meccanismo di pooling è del tutto trasparente per le applicazioni, come fa il connection manager a sapere quando entrare in azione?

Si dia uno sguardo alla seguente porzione di codice che utilizza un `DataSource` per accedere ai dati:

```
// Si ottiene una connessione da un pooled DataSource
// preliminarmente ottenuto in qualche maniera
Connection con = ds.getConnection("jdbc/mokadb", "mokauser", "mokaifx");


// Si eseguono un po' di operazioni SQL sul DB degli
// articoli, eventualmente disabilitando l'autocommit
...

// Chiude la connessione
con.close();
```

Quando si invoca il metodo `DataSource.getConnection()` non viene aperta una connessione come quando si invoca `DriverManager.getConnection()` ma, nel caso di pooled `DataSource`, viene restituito un riferimento a una connessione esistente e attiva nella cache o pool, evitando al database l'overhead di allocazione delle risorse e setup di nuova sessione. La connessione rimarrà allocata fin quando il client non termina il lavoro sul database.

Quando quindi nell'ultima istruzione il client decide di disconnettersi dal database o più propriamente decide di terminare la sessione di lavoro, la chiamata `Connection.close()` viene gestita dal pool manager che dealloca la connessione e la rende disponibile per altre applicazioni nella cache.

Transazioni distribuite

Il protocollo di transazione distribuita tra database XA esiste già da tempo. Dato che è un elemento ricorrente dell'enterprise computing JavaSoft ha pensato bene di includerlo nella sua piattaforma J2EE attraverso le due API JTA (Java Transaction API)e JTS (Java Transaction Service). Tale aggiunta non poteva non avere impatto ovviamente su JDBC: è stata così introdotta questa estensione standard per JDBC che permette di vedere il database come una risorsa abilitata alla transazione distribuita (posto che lo sia di suo).

Anche nel caso di transazioni distribuite il programmatore JDBC ha poco da scrivere, visto che per lui è trasparente il fatto che le operazioni SQL che la sua applicazione esegue facciano parte di una transazione distribuita. Più che altro in questo caso occorre sapere che vi sono delle restrizioni rispetto al caso di accesso a un solo database. Precisamente

bisogna evitare di chiamare i metodi `Connection.setAutoCommit()`, `Connection.commit()` e `Connection.rollback()`. E ciò perché questi metodi hanno direttamente impatto sulla connessione in corso che però viene gestita dal Transaction Manager del middle-tier server, che tipicamente è un application server.

Rowsets

Il rowset è un oggetto definito per rappresentare un set di righe di database. Un oggetto di questo tipo implementa l'interfaccia `javax.sql.RowSet` che a sua volta estende `java.sql.ResultSet`. L'interfaccia `RowSet` è stata disegnata con in mente il modello a componenti di JavaBeans.

In tal senso i rowsets hanno proprietà, a cui si accede con il tradizionale pattern get/set. Ad esempio:

```
// imposta la sorgente dati
rowset.setDataSourceName("jdbc/mokadb");
// definisce il livello di isolamento
rowset.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
// definisce query da eseguire
rowset.setCommand("SELECT NOME FROM ARTICOLISTI WHERE id=2");
// esegue il comando SQL
rowset.execute();
```

I rowset supportano come ogni JavaBean anche gli eventi: il meccanismo di gestione è lo stesso, permettendo a listener di registrarsi attraverso `RowSet.addRowSetListener()` e di ricevere, nel caso si verifichi un evento, una notifica e un oggetto `RowSetEvent` con informazioni sull'evento appena occorso.

Sun ha definito per i rowset solo delle interfacce che ne definiscono il modello e la logica di funzionamento (si tratta cioè una specifica). L'implementazione è lasciata al produttore del database, che definirà in oggetti che implementano queste interfacce la logica implementativa.

Sun ha anche individuato tre possibili e naturali implementazioni di rowset, che per ora non fanno parte della specifica per l'Optional Package: il `CachedRowSet`, il `JDBCRowSet`, il `WebRowSet`.

`CachedRowSet`

Il `CashedRowSet` è un contenitore tabulare di dati disconnesso, serializzabile e scrollabile. Queste proprietà lo rendono adatto a dispositivi disconnessi o solo occasionalmente connessi come i portatili, i palmari e altri dispositivi simili.

Un rowset così fatto permette a tali dispositivi di poter lavorare sui dati in locale senza connessione al server, come se si avesse una implementazione locale di JDBC e una locale

sorgente di dati, anche se in realtà i dati sono stati recuperati e immagazinati nel rowset in precedenza.

Per popolare un `CachedRowSet` prima di spedirlo, ad esempio, a un thin client che vi lavorerà, si utilizza del codice come il seguente:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM ARTICOLI");
CachedRowSet crset = new CachedRowSet();
crset.populate(rs);
```

Dopo aver popolato il rowset si può chiudere la connessione a cui si riferisce lo statement `stmt` (non mostrato) e spedire l'oggetto `crset` attraverso la rete a un client che può elaborarlo a piacimento prima di apportare le modifiche definitive al database. È ovvio qui il vantaggio per il client della possibilità di lavorare sui dati come se si avesse a disposizione un result set di JDBC 2, ma senza la necessità di una connessione aperta al database.

Il metodo alternativo per popolare il `CachedRowSet` è rappresentato da `CachedRowSet.execute()`, che esegue direttamente la query al database senza passare per un oggetto `ResultSet`.

Il `CachedRowSet`, dal punto di vista del client è in tutto e per tutto assimilabile a un `ResultSet`: l'applicazione client utilizzerà i metodi già visti per accedere e modificare i dati e solo quando si avrà la certezza di poter sottoporre le modifiche al database attraverso il metodo `CachedRowSet.acceptChanges()` il `CachedRowSet` aprirà la connessione al database utilizzando come parametri le proprietà del `CachedRowSet` stesso e, se i dati originali nel frattempo non sono cambiati (ma è una scelta dell'implementatore), le modifiche avranno luogo.

È possibile recuperare lo stato originale dei dati attraverso il metodo `CachedRowSet.restoreOriginal()`.

JDBCRowSet

Questo tipo di rowset è poco più di un layer intorno all'oggetto `ResultSet` con cui l'applicazione JDBC può accedere ai dati. Il vantaggio di questo wrapping è che il `JDBCRowSet` è di fatto un componente JavaBeans è quindi può essere inquadrato in un modello a componenti per la propria applicazione.

WebRowSet

Anche il `WebRowSet` è una specializzazione del `RowSet` che risolve un problema specifico.

Esso viene utilizzato per per trasferire i dati di un `ResultSet` a client fuori dal firewall aziendale, e che quindi possono al più utilizzare il protocollo HTTP per comuni-

care con il DB server. Questo rowset utilizza al proprio interno il protocollo HTTP per il tunnelling della connessione al database e per comunicare con un servlet sul Web Server, possibilmente utilizzando XML.

Java Beans

DI ANDREA GINI

La programmazione a componenti

Uno degli obiettivi più ambiziosi dell'ingegneria del software è organizzare lo sviluppo di sistemi in maniera simile a quanto è stato fatto in altre branche dell'ingegneria, dove la presenza di un mercato di parti standard altamente riutilizzabili permette di aumentare la produttività riducendo nel contempo i costi. Nella meccanica, ad esempio, esiste da tempo un importante mercato di componenti riutilizzabili, come viti, dadi, bulloni e ruote dentate; ciascuno di questi componenti trova facilmente posto in centinaia di prodotti diversi.

L'industria del software, sempre più orientata alla filosofia dei componenti, sta dando vita a due nuove figure di programmatore: il progettista di componenti e l'assemblatore di applicazioni.

Il primo ha il compito di scoprire e progettare oggetti software di uso comune, che possano essere utilizzati con successo in contesti differenti. Produttori in concorrenza tra di loro possono realizzare componenti compatibili, ma con caratteristiche prestazionali differenti. L'acquirente può orientarsi su un mercato che offre una pluralità di scelte e decidere in base al budget o a particolari esigenze di prestazione.

L'assemblatore di applicazioni, d'altra parte, è un professionista specializzato in un particolare dominio applicativo, capace di creare programmi complessi acquistando sul mercato componenti standard e combinandoli con strumenti grafici o linguaggi di scripting.

Questo capitolo offre un'analisi approfondita delle problematiche che si incontrano nella creazione di componenti in Java; attraverso gli esempi verrà comunque offerta una panoramica su come sia possibile assemblare applicazioni complesse a partire da componenti concepiti per il riuso.

La specifica Java Beans

JavaBeans è una specifica, ossia un insieme di regole seguendo le quali è possibile realizzare in Java componenti software riutilizzabili, che abbiano la capacità di interagire con altri componenti, realizzati da altri produttori, attraverso un protocollo di comunicazione comune.

Ogni Bean è caratterizzato dai servizi che è in grado di offrire e può essere utilizzato in un ambiente di sviluppo differente rispetto a quello in cui è stato realizzato. Quest'ultimo punto è cruciale nella filosofia dei componenti: sebbene i Java Beans siano a tutti gli effetti classi Java, e possano essere manipolati completamente per via programmatica, essi vengono spesso utilizzati in ambienti di sviluppo diversi, come tool grafici o linguaggi di scripting.

I tool grafici, tipo JBuilder, permettono di manipolare i componenti in maniera visuale. Un assemblatore di componenti può selezionare i Beans da una palette, inserirli in un apposito contenitore, impostarne le proprietà, collegare gli eventi di un Bean ai metodi di un altro, generando in tal modo applicazioni, Applet, Servlet e persino nuovi componenti senza scrivere una sola riga di codice.

I linguaggi di scripting, di contro, offrono una maggiore flessibilità rispetto ai tool grafici, senza presentare le complicazioni di un linguaggio generico. La programmazione di pagine web dinamiche, uno dei domini applicativi di maggior attualità, deve il suo rapido sviluppo a un'intelligente politica di stratificazione, che vede le funzionalità di più basso livello, come la gestione dei database, la Businnes Logic o l'interfacciamento con le risorse di sistema, incapsulate all'interno di JavaBeans, mentre tutto l'aspetto della presentazione viene sviluppato con un semplice linguaggio di scripting, tipo Java Server Pages o PHP.

Il modello a componenti Java Beans

Un modello a componenti è caratterizzato da almeno sette fattori: proprietà, metodi, introspezione, personalizzazione, persistenza, eventi e modalità di deployment. Nei prossimi paragrafi si analizzerà il ruolo di ciascuno di questi aspetti all'interno della specifica Java Beans; quindi si procederà a descriverne l'implementazione in Java.

Proprietà

Le proprietà sono attributi privati, accessibili solamente attraverso appositi metodi `get` e `set`. Tali metodi costituiscono l'unica via di accesso pubblica alle proprietà, cosa che permette al progettista di componenti di stabilire per ogni parametro precise regole di accesso. Se si utilizzano i Bean all'interno di un programma di sviluppo visuale, le proprietà di un componente vengono visualizzate in un apposito pannello, che permette di modificarne il valore con un opportuno strumento grafico.

Metodi

I metodi di un Bean sono metodi pubblici Java, con l'unica differenza che essi risultano accessibili anche attraverso linguaggi di scripting e Builder Tools. I metodi sono la prima e più importante via d'accesso ai servizi di un Bean.

Introspezione

I Builder Tools scoprono i servizi di un Bean (proprietà, metodi ed eventi) attraverso un processo noto come introspezione, che consiste principalmente nell'interrogare il componente per conoscerne i metodi, e dedurre da questi le caratteristiche. Il progettista di componenti può attivare l'introspezione in due maniere: seguendo precise convenzioni nella formulazione delle firme dei metodi, o creando una speciale classe `BeanInfo`, che fornisce un elenco esplicito dei servizi di un particolare Bean.

La prima via è senza dubbio la più semplice: se si definiscono i metodi di accesso a un determinato servizio seguendo le regole di naming descritte dalla specifica JavaBeans, i tool grafici saranno in grado, grazie alla reflection, di individuare i servizi di un Bean semplicemente osservandone l'interfaccia di programmazione. Il ricorso ai BeanInfo, d'altro canto, torna utile in tutti quei casi in cui sia necessario mascherare alcuni metodi, in modo da esporre solamente un sottoinsieme dei servizi effettivi del Bean.

Personalizzazione

Durante il lavoro di composizione di Java Beans all'interno di un tool grafico, un apposito Property Sheet, generato al volo dal programma di composizione, mostra lo stato delle proprietà e permette di modificarle con un opportuno strumento grafico, tipo un campo di testo per valori `String` o una palette per proprietà `Color`. Simili strumenti grafici vengono detti editor di proprietà.

I tool grafici dispongono di editor di proprietà in grado di supportare i tipi Java più comuni, come i tipi numerici, le stringhe e i colori; nel caso si desideri rendere editabile una proprietà di un tipo diverso, è necessario realizzare un'opportuna classe di supporto, conforme all'interfaccia `PropertyEditor`. Quando invece si desideri fornire un controllo totale sulla configurazione di un Bean, è possibile definire un Bean Customizer, una speciale applicazione grafica specializzata nella configurazione di un particolare tipo di componenti.

Persistenza

La persistenza permette ad un Bean di salvare il proprio stato e di ripristinarlo in un secondo tempo. JavaBeans supporta la persistenza grazie all'Object Serialization, che permette di risolvere questo problema in modo molto rapido.

Eventi

Nella programmazione a oggetti tradizionale non esiste nessuna convenzione su come modellare lo scambio di messaggi tra oggetti. Ogni programmatore adotta un proprio sistema, creando una fitta rete di dipendenze che rende molto difficile il riutilizzo di oggetti in contesti differenti da quello di partenza. Gli oggetti Java progettati secondo la specifica Java Beans adottano un meccanismo di comunicazione basato sugli eventi, simile a quello utilizzato nei componenti grafici Swing e AWT. L'esistenza di un unico protocollo di comunicazione standard garantisce l'intercomunicabilità tra componenti, indipendentemente da chi li abbia prodotti.

Deployment

I JavaBeans possono essere consegnati, in gruppo o singolarmente, attraverso file JAR, speciali archivi compressi in grado di trasportare tutto quello di cui un Bean ha bisogno, come classi, immagini o altri file di supporto. Grazie ai file `.jar` è possibile consegnare i Beans con una modalità del tipo "chiavi in mano": l'acquirente deve solamente caricare un file JAR nel proprio ambiente di sviluppo e i Beans in esso contenuti verranno subito messi a disposizione. L'impacchettamento di classi Java all'interno di file JAR segue poche semplici regole, che verranno descritte negli esempi del capitolo.

Guida all'implementazione dei JavaBeans

Realizzare un componente Java Bean è un compito alla portata di qualunque programmatore Java che disponga di buone conoscenze di sviluppo Object Oriented. Nei paragrafi seguenti verranno descritte dettagliatamente le convenzioni di naming dettate dalla specifica, e verranno fornite le istruzioni su come scrivere le poche righe di codice necessarie a implementare i meccanismi che caratterizzano i servizi Bean. Infine verranno presentati degli esempi, che permetteranno di impratichirsi con il processo di implementazione delle specifiche.

Le proprietà

Le proprietà sono attributi che descrivono l'aspetto e il comportamento di un Bean, e che possono essere modificate durante tutto il ciclo di vita del componente. Di base, le proprietà sono attributi privati, ai quali si accede attraverso una coppia di metodi della forma:

```
public <PropertyType> get<PropertyName>()

public void set<PropertyName>(<PropertyType> property)
```

La convenzione di aggiungere il prefisso `get` e `set` ai metodi che forniscono l'accesso

a una proprietà, permette ad esempio ai tool grafici di rilevare le proprietà Bean, determinarne le regole di accesso (Read Only o Read/Write), dedurne il tipo, visualizzare le proprietà su un apposito Property Sheet e individuare l'editor di proprietà più adatto al caso.

Se ad esempio un tool grafico scopre, grazie all'introspezione, la coppia di metodi

```
public Color getForegroundColor() { ... }
public void setForegroundColor(Color c) { ... }
```

da questi conclude che esiste una proprietà chiamata `foregroundColor` (notare la prima lettera minuscola), accessibile sia in lettura che in scrittura, di tipo `Color`. A questo punto, il tool può cercare un editor di proprietà per parametri di tipo `Color`, e mostrare la proprietà su un property sheet in modo che possa essere vista e manipolata dal programmatore.

Proprietà indicizzate (Indexed Property)

Le proprietà indicizzate permettono di gestire collezioni di valori accessibili attraverso indice, in maniera simile a come si fa con un vettore. Lo schema di composizione dei metodi di accesso di una proprietà indicizzata è il seguente:

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

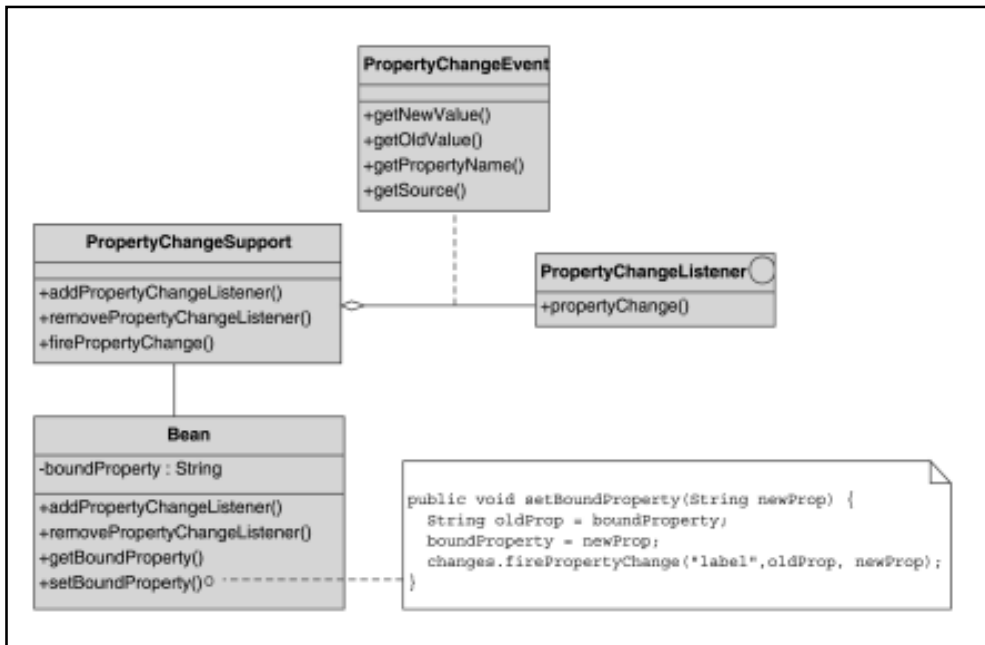
per i metodi che permettono di manipolare l'intera collection, mentre per accedere ai singoli elementi, si deve predisporre una coppia di metodi del tipo:

```
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> value);
```

Proprietà bound

Le proprietà semplici, così come sono state descritte nei paragrafi precedenti, seguono una convenzione radicata da tempo nella normale programmazione a oggetti. Le proprietà bound, al contrario, sono caratteristiche dell'universo dei componenti, dove si verifica molto spesso la necessità di collegare il valore delle proprietà di un componente a quelli di un'altro, in modo tale che si mantengano aggiornati. I metodi `set` delle proprietà bound, inviano una notifica a tutti gli ascoltatori registrati ogni qualvolta viene alterato il valore della proprietà. Il meccanismo di ascolto–notifica, simile a quello degli eventi Swing e AWT, segue il pattern Observer.

Le proprietà bound, a differenza degli eventi Swing, utilizzano un unico tipo di evento, `ChangeEvent`, cosa che semplifica il processo di implementazione. La classe `PropertyChangeSupport`, presente all'interno del package `java.bean`, fornisce i metodi che gestiscono la lista degli ascoltatori e quelli che producono l'invio degli eventi.

Figura 8.1 – *Il meccanismo di notifica di eventi bound segue il pattern Observer*

Un oggetto che voglia mettersi in ascolto di una proprietà, deve implementare l'interfaccia `PropertyChangeListener` e deve registrarsi presso la sorgente di eventi. L'oggetto `PropertyChangeEvent` incapsula le informazioni riguardo alla proprietà modificata, alla sorgente e al valore della proprietà.

Come implementare il supporto alle proprietà bound

Per aggiungere a un Bean il supporto alle proprietà bound, bisogna anzitutto importare il package `java.beans.*`, in modo da garantire l'accesso alle classi `PropertyChangeSupport` e `PropertyChangeEvent`. Quindi bisogna creare un oggetto `PropertyChangeSupport`, che ha il compito di mantenere la lista degli ascoltatori e di fornire i metodi che gestiscono l'invio degli eventi.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

A questo punto bisogna realizzare, nella propria classe, i metodi che permettono di gestire la lista degli ascoltatori. Tali metodi sono dei semplici metodi Wrapper che fanno

riferimento a metodi con la stessa firma, presenti nel `PropertyChangeSupport`:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
   changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l) {
   changes.removePropertyChangeListener(l);
}
```

La presenza dei metodi `addPropertyChangeListener()` e `removePropertyChangeListener()` permette ai tool grafici di riconoscere un oggetto in grado di inviare proprietà bound e di mettere a disposizione un'opportuna voce nel menù di gestione degli eventi.

L'ultimo passaggio consiste nel modificare i metodi `set` relativi alle proprietà che si vuole rendere bound, per fare in modo che venga generato un `PropertyChangeEvent` ogni volta che la proprietà viene reimpostata

```
public void setColor(Color newColor) {
   Color oldColor = color;
   color = newColor;
   changes.firePropertyChange("color", oldColor , newColor);
}
```

Nel caso di proprietà read only, prive di metodo `set`, l'invio dell'evento dovrà avvenire all'interno del metodo che attua la modifica della proprietà. Un aspetto interessante del meccanismo di invio di `PropertyChangeEvent`, è che essi trasportano sia il nuovo valore che quello vecchio. Questa scelta dispensa chi implementa un ascoltatore dal compito di mantenere una copia del valore, qualora questo fosse necessario, dal momento che l'evento viene propagato *dopo* la modifica della relativa proprietà. Il metodo `fireChangeEvent()` della classe `PropertyChangeListener` fornisce il servizio di Event Dispatching:

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

In pratica esso impacchetta i parametri in un oggetto `PropertyChangeEvent`, e chiama il metodo `propertyChange(PropertyChangeEvent p)` su tutti gli ascoltatori registrati. I parametri vengono trattati come Object, e nel caso si debbano inviare proprietà espresse in termini di tipi primitivi, occorre incapsularle nell'opportuno Wrapper (`Integer` per valori `int`, `Double` per valori `double` e così via). Per facilitare questo compito, la classe `propertyChangeSupport` prevede delle varianti di `firePropertyChange` per valori `int` e `boolean`.

Come implementare il supporto alle proprietà bound su sottoclassi di `JComponent`

La classe `JComponent`, superclasse di tutti i componenti Swing, dispone del supporto nativo alla gestione di proprietà bound. Di base essa fornisce i metodi `addPropertyChangeListener` e `removePropertyChangeListener`, oltre a una collezione di metodi `firePropertyChange` adatta ad ogni tipo primitivo. In questo caso l'implementazione di una proprietà bound richiederà solo una modifica al metodo `set` preposto, similmente a come descritto nell'ultimo passaggio del precedente paragrafo, con la differenza che non è necessario ricorrere a un oggetto `propertyChangeSupport` per inviare la proprietà:

```
public void setColor(Color newColor) {
   Color oldColor = color;
   color = newColor;
   firePropertyChange("color", oldColor , newColor);
}
```

Ascoltatori di proprietà

Se si desidera mettersi in ascolto di una proprietà, occorre definire un opportuno oggetto `PropertyChangeListener` e registrarlo presso il Bean. Un `PropertyChangeListener` deve definire il metodo `propertyChange(PropertyChangeEvent e)`, che viene chiamato quando avviene la modifica di una proprietà bound.

Un `PropertyChangeListener` viene notificato quando avviene la modifica di *una qualunque* proprietà bound: per questa ragione esso deve, come prima cosa, verificare, che la proprietà appena modificata sia quella alla quale si è interessati. Una simile verifica richiede una chiamata al metodo `getPropertyName` di `PropertyChangeEvent`, che restituisce il nome della proprietà. Per convenzione, i nomi di proprietà vengono estratti dai nomi dichiarati nei metodi `get` e `set`, con la prima lettera minuscola. Il seguente frammento di codice presenta un tipico `PropertyChangeListener`, che ascolta la proprietà `foregroundColor`:

```
public class Listener implements PropertyChangeListener() {
   public void propertyChange(PropertyChangeEvent e) {
      if(e.getPropertyName().equals("foregroundColor"))
         System.out.println(e.getNewValue());
   }
}
```

Un esempio di Bean con proprietà bound

Un Java Bean rappresenta un mattone di un programma. Ogni componente è un'unità di utilizzo abbastanza grossa da incorporare una funzionalità evoluta, ma piccola rispetto

ad un programma fatto e finito. Il concetto del riuso può essere presente a diversi livelli del progetto: il seguente Bean fornisce un esempio di elevata versatilità

Il Bean `PhotoAlbum` è un pannello grafico al cui interno vengono caricate delle immagini. Il metodo `showNext()` permette di passare da un'immagine all'altra, in modo ciclico. Il numero ed il tipo di immagini viene determinato al momento dell'avvio: durante la fase di costruzione viene letto il file `comment.txt`, presente nella directory `images`, che contiene una riga di commento per ogni immagine presente nella cartella. Le immagini devono essere nominate in modo progressivo (`img0.jpg`, `img1.jpg`, `img2.jpg`…) e devono essere presenti in numero uguale alle righe del file `comment.txt`. Questa scelta progettuale consente di introdurre il riuso a un livello abbastanza alto: qualunque utente, anche con scarse conoscenze del linguaggio, può personalizzare il componente, inserendo le sue foto preferite, senza la necessità di alterare il codice sorgente.

Il Bean `PhotoAlbum` ha tre proprietà:

- `imageNumber`, che restituisce il numero di immagini contenute nell'album. Essendo una quantità immutabile, tale proprietà è stata implementata come proprietà semplice.

- `imageIndex`: restituisce l'indice dell'immagine attualmente visualizzata. Al cambio di immagine viene inviato un `PropertyChangeEvent`.

- `imageComment`: restituisce una stringa di commento all'immagine. Anche in questo caso, al cambio di immagine viene generato un `PropertyChangeEvent`.

Il Bean viene definito come sottoclasse di `JPanel`: per questo motivo non vengono dichiarati i metodi `addPropertyChangeListener` e `removePropertyChangeListener`, già presenti nella superclasse. L'invio delle proprietà verrà messo in atto grazie al metodo `firePropertyChange` di `JComponent`.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class PhotoAlbum extends JPanel {

   private Vector comments = new Vector();
   private int imageIndex;
```

```
   public PhotoAlbum() {
      super();
      setLayout(new BorderLayout());
      setupComments();
      imageIndex = 0;
      showNext();
   }
   private void setupComments() {
      try {
         URL indexUrl = getClass().getResource("images/" + "comments.txt");
         InputStream in = indexUrl.openStream();
         BufferedReader lineReader
         = new BufferedReader(new InputStreamReader(in));
         String line;
         while((line = lineReader.readLine())!=null)
            comments.add(line);
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
   public int getImageNumber() {
      return comments.size();
   }
   public int getImageIndex() {
      return imageIndex;
   }
   public String getImageComment() {
      return (String)comments.elementAt(imageIndex);
   }
   public void showNext() {
      int oldImageIndex = imageIndex;
      imageIndex = ((imageIndex +1 ) % comments.size());
      String imageName = "img" + Integer.toString(imageIndex) + ".jpg";
      showImage(getClass().getResource("images/" + imageName));
      String oldImageComment = (String)comments.elementAt(oldImageIndex);
      String currentImageComment = (String)comments.elementAt(imageIndex);
      firePropertyChange("imageComment", oldImageComment, currentImageComment);
      firePropertyChange("imageIndex", oldImageIndex, imageIndex);
   }
   private void showImage(URL imageUrl) {
      ImageIcon img = new ImageIcon(imageUrl);
      JLabel picture = new JLabel(img);
      JScrollPane pictureScrollPane = new JScrollPane(picture);
      removeAll();
      add(BorderLayout.CENTER,pictureScrollPane);
      validate();
   }
}
```

È possibile testare il Bean come fosse una normale classe Java, utilizzando queste semplici righe di codice:

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import com.mokabyte.mokabook.javaBeans.photoAlbum.*;
import java.beans.*;
import javax.swing.*;

public class PhotoAlbumTest {
public static void main(String argv[]) {
   JFrame f = new JFrame("Photo Album");
   PhotoAlbum p = new PhotoAlbum();
   f.getContentPane().add(p);
   p.addPropertyChangeListener(new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent e) {
         System.out.println(e.getPropertyName() + ": " + e.getNewValue());
      }
   });
   f.setSize(500,400);
   f.setVisible(true);

   while(true)
      for(int i=0;i<7;i++) {
         p.showNext();
         try {Thread.sleep(1000);}catch(Exception e) {}
      }
}
}
```

Figura 8.2 – *Un programma di prova per il Bean* `PhotoAlbum`

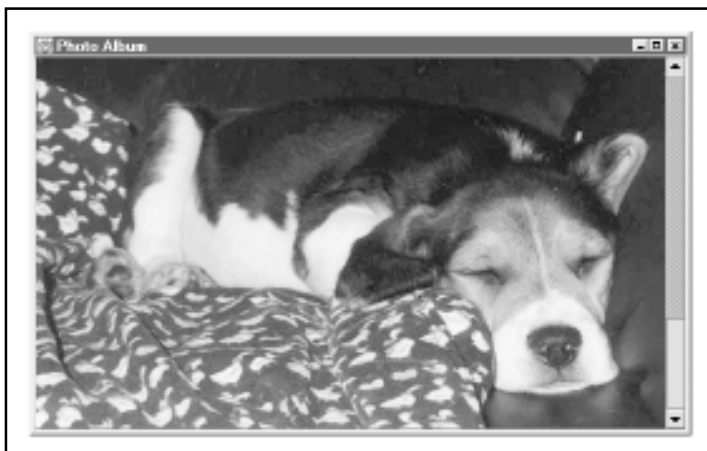
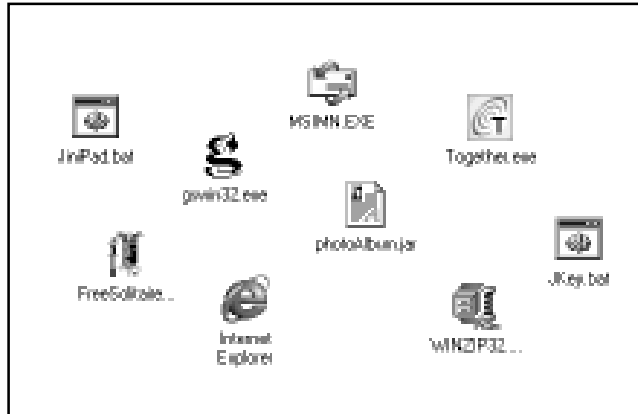



Figura 8.3 – *Un file JAR opportunamente confezionato può essere aperto con un opportuno tool come Jar o WinZip*

Creazione di un file JAR

Prima di procedere alla consegna del Bean entro un file JAR, bisogna anzitutto compilare le classi `PhotoAlbum.java` e `PhotoAlbumTest.java`, che devono trovarsi nella cartella `com\mokabyte\mokabook\javaBeans\`

```
javac com\mokabyte\mokabook\javaBeans\photoAlbum\*.java
```

A questo punto bisogna creare, ricorrendo a un semplice editor di testo tipo Notepad, un file `photoAlbumManifest.tmp` con il seguente contenuto

```
Main-Class: com.mokabyte.mokabook.javaBeans.photoAlbum.PhotoAlbumTest
Name: com/mokabyte/mokabook/javaBeans/photoAlbum/PhotoAlbum.class
Java-Bean: True
```

Le prime due righe, opzionali, segnalano la presenza di una classe dotata di metodo main.

Le ultime due righe del file manifest specificano che la classe `PhotoAlbum.class` è un Java Bean. Se l'archivio contiene più di un Bean, è necessario elencarli tutti.

Per generare l'archivio `photoAlbum.jar`, bisogna digitare la riga di comando:

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp
   com\mokabyte\mokabook\javaBeans\photoAlbum\*.class
   com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

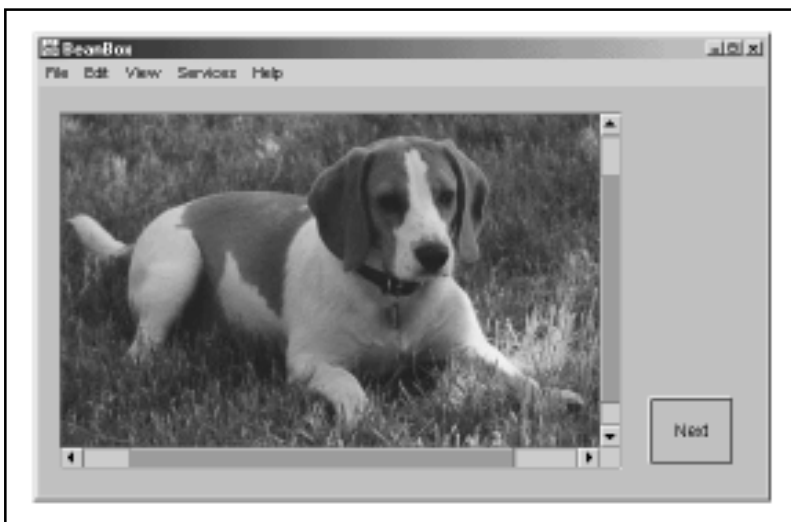
Il file così generato contiene tutte le classi e le immagini necessarie a dar vita al Bean `PhotoAlbum`. Tale file potrà essere utilizzato facilmente all'interno di tool grafici o di pagine web, racchiuso dentro una Applet.
Il file `.jar` potrà essere avviato digitando

```
java PhotoAlbum.jar
```

Le istruzioni fornite sono valide per la piattaforma Windows. Su piattaforma Unix, le eventuali occorrenze del simbolo "\", che funge da path separator su piattaforme Windows, andranno sostituite col simbolo "/". Le convenzioni adottate all'interno del file manifest valgono invece su entrambe le piattaforme.

Integrazione con altri Beans

Nonostante il Bean `PhotoAlbum` fornisca un servizio abbastanza evoluto, non è ancora classificabile come applicazione. Esso, opportunamente integrato con altri Beans, può comunque dar vita a numerosi programmi; di seguito, ecco qualche esempio: collegato a un `CalendarBean`, `PhotoAlbum` può dar vita a un simpatico calendario elettronico;

Figura 8.4 – *Combinando, all'interno del Bean Box, il Bean* `PhotoAlbum` *con un pulsante Bean, si ottiene una piccola applicazione*

collegando un bottone Bean al metodo `showNext()` è possibile creare un album interattivo, impacchettarlo su un'Applet e pubblicarlo su Internet; impacchettando il Bean `PhotoAlbum` con foto natalizie, e collegandolo con un Bean Carillon, si può ottenere un biglietto di auguri elettronico.

A questi esempi se ne possono facilmente aggiungere altri; altri ancora diventano possibili aggiungendo al Bean nuovi metodi, come `previousImage()` e `setImageAt(int i)`; un compito ormai alla portata del lettore che fornisce un ottimo pretesto per esercitarsi.

Eventi Bean

La notifica del cambiamento di valore delle proprietà bound è un meccanismo di comunicazione tra Beans. Se si vuole che un Bean sia in grado di propagare eventi di tipo più generico, o comunque eventi che non è comodo rappresentare come un cambiamento di stato, è possibile utilizzare un meccanismo di eventi generico, del tutto simile a quello presente nei componenti grafici Swing e AWT. I prossimi paragrafi servono a illustrare le tre fasi dell'implementazione: creazione dell'evento, definizione dell'ascoltatore e infine creazione della sorgente di eventi.

Creazione di un evento

Per implementare un meccanismo di comunicazione basato su eventi, occorre anzitutto definire un'opportuna sottoclasse di `EventObject`, che racchiuda tutte le informazioni relative all'evento da propagare.

```
public class <EventType> extends EventObject {
   private <ParamType> param
   public <EventType>(Object source,<ParamType> param) {
      super(source);
      this.param = param;
   }
   public <ParamType> getParameter() {
      return param;
   }
}
```

La principale variazione sul tema si ha sul numero e sul tipo di parametri: tanto più complesso è l'evento da descrivere, maggiori saranno i parametri in gioco. L'unico parametro che è obbligatorio fornire è un reference all'oggetto che ha generato l'evento: tale reference, richiamabile con il metodo `getSource()` della classe `EventObject`, permetterà all'ascoltatore di interrogare la sorgente degli eventi qualora ce ne fosse bisogno.

Destinatari di eventi

Il secondo passaggio è quello di definire l'interfaccia di programmazione degli ascoltatori di eventi. Tale interfaccia deve essere definita come sottoclasse di `EventListener`, per essere riconoscibile come ascoltatore dall'`Introspector`. Lo schema di sviluppo degli ascoltatori segue lo schema

```
import java.awt.event.*;

public Interface <EventListener> extends EventListener {
   public void <EventType>Performed(<EventType> e);
}
```

Le convenzioni di naming dei metodi dell'interfaccia non seguono uno schema standard: la convenzione descritta nell'esempio, `<EventType>performed`, può essere seguita o meno. L'importante è che il nome dei metodi dell'interfaccia `Listener` suggeriscano il tipo di azione sottostante, e che accettino come parametro un evento del tipo giusto.

Sorgenti di eventi

Se si desidera aggiungere a un Bean la capacità di generare eventi, occorre implementare una coppia di metodi

```
add<EventListenerType>(<EventListenerType> l)

remove<EventListenerType>(<EventListenerType> l)
```

La gestione della lista degli ascoltatori e l'invio degli eventi segue una formula standard, descritta nelle righe seguenti:

```
private Vector listeners = new Vector();

public void add<EventListenerType>(<EventListenerType> l) {
   listeners.add(l);
}
public void remove<EventListenerType>(<EventListenerType> l) {
   listeners.remove(l);
}
protected void fire<Eventype>(<EvenType> e) {
   Enumeration listenersEnumeration = listeners.elements();
   while(listenersEnumeration.hasMoreElements()) {
   <EventListenerType> listener
   = (<EventListenerType>)listenersEnumeration.nextElement();
      listener.<EventType>Performed(e);
   }
}
```

Sorgenti unicast

In alcuni casi occorre definire sorgenti di eventi capaci di servire un unico ascoltatore. Per implementare tali classi, che fungono da sorgenti unicast, si può seguire il seguente modello

```
private <EventListenerType> listener;

public void add<EventListenerType>(
<EventListenerType> l) throws TooManyListenersException {
   if(listener == null)
      listener = l;
   else
      throw new TooManyListenerException();
}
public void remove<EventListenerType>(<EventListenerType> l) {
   listener = null;
}
protected void fire<Eventype>(<EvenType> e) {
   if(listener! = null)
      listener.<EventType>Performed(e);
}
```

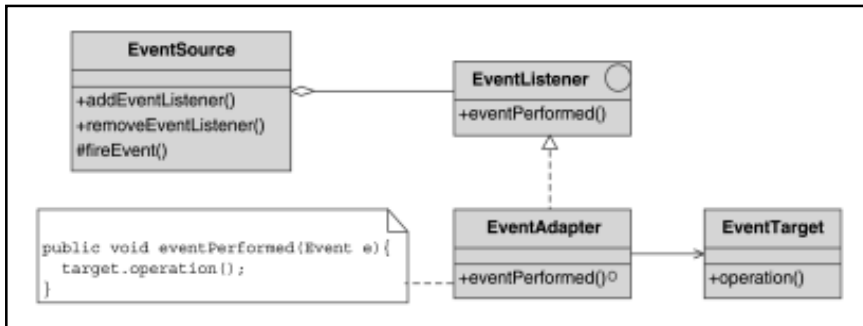
Ascoltatori di eventi: `Event Adapter`

Se si vuole che un evento generato da un Bean scateni un'azione su un altro Bean, è necessario creare un oggetto che realizzi un collegamento tra i due. Tale classe, detta Adapter, viene registrata come ascoltatore presso la sorgente dell'evento, e formula una chiamata al metodo destinazione ogni volta che riceve una notifica dal Bean sorgente.

Gli strumenti grafici tipo JBuilder generano questo tipo di classi in maniera automatica: tutto quello che l'utente deve fare è collegare, con pochi click di mouse, l'evento di un Bean sorgente a un metodo di un Bean target. Qui di seguito viene riportato il codice di un Adapter, generato automaticamente dal Bean Box, che collega la pressione di un pulsante al metodo `startJuggling(ActionEvent e)` del Bean `Juggler`.

```
// Automatically generated event hookup file.

public class ___Hookup_172935aa26 implements java.awt.event.ActionListener,
                                             java.io.Serializable {

   public void setTarget(sunw.demo.juggler.Juggler t) {
      target = t;
   }

   public void actionPerformed(java.awt.event.ActionEvent arg0) {
      target.startJuggling(arg0);
```

```
   }

   private sunw.demo.juggler.Juggler target;
}
```

Figura 8.5 – *Un Adapter funge da ponte di collegamento tra gli eventi di un Bean e i metodi di un altro*

Un esempio di Bean con eventi

Il prossimo esempio è un Bean `Timer`, che ha il compito di generare battiti di orologio a intervalli regolari. Questo componente è un tipico esempio di Bean non grafico.

La prima classe che si definisce è quella che implementa il tipo di evento

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerEvent extends EventObject implements Serializable {

   public TimerEvent(Object source) {
      super(source);
   }
}
```

Come si può vedere, l'implementazione di un nuovo tipo di evento è questione di poche righe di codice. L'unico particolare degno di nota è che il costruttore del nuovo tipo di evento deve invocare il costruttore della superclasse, passando un reference alla sorgente dell'evento.

L'interfaccia che rappresenta l'ascoltatore deve estendere l'interfaccia `EventListener`; a parte questo, al suo interno si può definire un numero arbitrario di metodi, la cui unica costante è quella di avere come parametro un reference all'evento da propagare.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public interface TimerListener extends java.util.EventListener {
   public void clockTicked(TimerEvent e);
}
```

Per finire, ecco il Bean vero e proprio. Come si può notare, esso implementa l'interfaccia `Serializable` che rende possibile la serializzazione.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerBean implements Serializable {

private int time = 1000;
private transient TimerThread timerThread;
private Vector timerListeners = new Vector();

public void addTimerListener(TimerListener t) {
   timerListeners.add(t);
}
public void removeTimerListener(TimerListener t) {
   timerListeners.remove(t);
}
protected void fireTimerEvent(TimerEvent e) {
   Enumeration listeners = timerListeners.elements();
   while(listeners.hasMoreElements())
      ((TimerListener)listeners.nextElement()).clockTicked(e);
}
public synchronized void setMillis(int millis) {
   time = millis;
}
public synchronized int getMillis() {
   return time;
}
public synchronized void startTimer() {
   if(timerThread!=null)
      forceTick();
```

```
   timerThread = new TimerThread();
   timerThread.start();
}
public synchronized void stopTimer() {
   if(timerThread == null)
      return;

   timerThread.killTimer();
   timerThread = null;
}
public synchronized void forceTick() {
   if(timerThread!=null) {
      stopTimer();
      startTimer();
   }
   else
      fireTimerEvent(new TimerEvent(this));
}

   class TimerThread extends Thread {
      private boolean running = true;

      public synchronized void killTimer() {
         running = false;
      }
      private synchronized boolean isRunning() {
         return running;
      }
      public void run() {
         while(true)
            try {
               if(isRunning()) {
                  fireTimerEvent(new TimerEvent(TimerBean.this));
                  Thread.sleep(getMillis());
               }
               else
                  break;
         catch(InterruptedException e) {}
         }
      }
   }
}
```

I primi tre metodi servono a gestire la lista degli ascoltatori. Il terzo e il quarto gestiscono la proprietà `millis`, ossia il tempo, in millisecondi, tra un tick e l'altro. I due metodi successivi, `startTimer`, `stopTimer`, servono ad avviare e fermare il timer, mentre `forceTick` lancia un tick e riavvia il timer, se questo è attivo. Il timer vero e proprio

viene implementato grazie a una classe interna `TimerThread`, sottoclasse di `Thread`. Si noti il metodo `killTimer`, che permette di terminare in modo pulito la vita del thread: questa soluzione è da preferire al metodo `stop` (deprecato a partire dal JDK 1.1), che in certi casi può provocare la terminazione del thread in uno stato inconsistente.

Per compilare le classi del Bean, bisogna usare la seguente riga di comando

```
javac com\mokabyte\mokabook\javaBeans\timer\*.java
```

Per impacchettare il Bean in un file `.jar`, è necessario per prima cosa creare con un editor di testo il file `timerManifest.tmp`, con le seguenti righe

```
Name: com/mokabyte/mokabook/javaBeans/timer/TimerBean.class
Java-Bean: True
```

Per creare l'archivio si deve quindi digitare il seguente comando

```
jar cfm timer.jar timerManifest.tmp com\mokabyte\mokabook\javaBeans\timer\*.class
```

Per testare la classe `TimerBean`, si può usare il seguente programma, che crea un oggetto `TimerBean` e registra un `TimerListener` il quale stampa a video una scritta ad ogni tick del timer.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public class TimerTest {
   public static void main(String argv[]) {
      TimerBean t = new TimerBean();
      t.addTimerListener(new TimerListener() {
         public void clockTicked(TimerEvent e) {
            System.out.println("Tick");
         }
      });
      t.startTimer();
   }
}
```

Introspezione: l'interfaccia `BeanInfo`

Le convenzioni di naming descritte nei paragrafi precedenti permettono ai tool grafici abilitati ai Beans di scoprire i servizi di un componente grazie alla reflection. Questo processo automatico è certamente comodo, ma ha il difetto di non offrire nessun tipo di

controllo sul numero e sul tipo di servizi da mostrare. In alcune occasioni può essere necessario mascherare un certo numero di servizi, specie quelli ereditati da una superclasse.

I Beans creati a partire dalla classe `JComponent`, ad esempio, ereditano automaticamente più di dieci attributi (dimensioni, colore, allineamento…) e ben dodici tipi diversi di evento (`ComponentEvent`, `MouseEvent`, `HierarchyEvent`…). Un simile eccesso provoca di solito disorientamento nell'utente; in questi casi è preferibile fornire un elenco esplicito dei servizi da associare al nostro Bean, in modo da "ripulire" gli eccessi.

Per raggiungere questo obiettivo, bisogna associare al Bean una classe di supporto, che implementi l'interfaccia `BeanInfo`. Una classe `BeanInfo` permette di fare un certo numero di cose: esporre solamente i servizi che si desidera rendere visibili, aggirare le convenzioni di naming imposte dalle specifiche Java Beans, associare al Bean un'icona e attribuire ai servizi nomi più descrittivi di quelli rilevabili con il processo di analisi delle firme dei metodi.

Creare una classe `BeanInfo`

Per creare una classe `BeanInfo` bisogna anzitutto definire una classe con lo stesso nome del Bean, a cui si deve aggiungere il suffisso `BeanInfo`. Per semplificare il lavoro si può estendere `SimpleBeanInfo`, una classe che fornisce un'implementazione nulla di tutti i metodi dell'interfaccia. In questo modo ci si limiterà a sovrascrivere solamente i metodi che interessano, lasciando tutti gli altri con l'impostazione di default.

Per ridefinire il numero ed il tipo dei servizi Bean, occorre agire in modo appropriato a restituire le proprietà, i metodi o gli eventi che si desidera esporre. Opzionalmente, si può associare un'icona al Bean, definendo il metodo `public java.awt.Image getIcon(int iconKind)`. Per finire, si può specificare la classe del Bean e il suo Customizer, qualora ne esista uno, con il metodo `public BeanDescriptor getBeanDescriptor()`.

La classe `BeanInfo` così prodotta deve essere messa nello stesso package che contiene il Bean. In assenza di una classe `BeanInfo`, i servizi di un Bean vengono trovati con la reflection.

Feature Descriptors

Una classe di tipo `BeanInfo` restituisce, tramite i seguenti metodi, vettori di *descriptors* che contengono informazioni relative ad ogni proprietà, metodo o evento che il progettista di un Bean desidera esporre.

```
PropertyDescriptor[] getPropertyDescriptors();
MethodDescriptor[] getMethodDescriptors();
EventSetDescriptor[] getEventSetDescriptors();
```

Ogni Descriptor fornisce una precisa rappresentazione di una classe di servizi Bean. Il package `java.bean` implementa le seguenti classi:

- `FeatureDescriptor`: è la classe base per tutte le altre classi Descriptor, e definisce gli aspetti comuni a tutta la famiglia.

- `BeanDescriptor`: descrive il tipo e il nome della classe Bean associati, oltre a fornire il Customizer, se ne esiste uno.

- `PropertyDescriptor`: descrive le proprietà del Bean.

- `IndexedPropertyDescriptor`: è una sottoclasse di `PropertyDescriptor`, e descrive le proprietà indicizzate.

- `EventSetDescriptor`: descrive gli eventi che il Bean è in grado di inviare.

- `MethodDescriptor`: descrive i metodi del Bean.

- `ParameterDescriptor`: descrive i parametri dei metodi.

Esempio

In questo esempio si analizzerà un `BeanInfo` per il Bean `PhotoAlbum`, che permette di nascondere una grossa quantità di servizi Bean che per default vengono ereditati dalla superclasse `JPanel`.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.beans.*;
import com.mokabyte.mokabook.javaBeans.photoAlbum.*;

public class PhotoAlbumBeanInfo extends SimpleBeanInfo {

private static final Class beanClass = PhotoAlbum.class;

   public PropertyDescriptor[] getPropertyDescriptors() {
      try {
         PropertyDescriptor imageNumber
         = new PropertyDescriptor("imageNumber", beanClass,
                                   "getImageNumber", null);
         PropertyDescriptor imageIndex
         = new PropertyDescriptor("imageIndex", beanClass,
                                   "getImageIndex", null);
```

```
         PropertyDescriptor imageComment
         = new PropertyDescriptor("imageComment", beanClass,
                                  "getImageComment", null);

         imageIndex.setBound(true);
         imageComment.setBound(true);

         PropertyDescriptor properties[]
         = {imageNumber, imageIndex, imageComment};
         return properties;
      } catch (IntrospectionException e) {
          throw new Error(e.toString());
      }
   }
   public EventSetDescriptor[] getEventSetDescriptors() {
      try {
         EventSetDescriptor changed
         = new EventSetDescriptor(beanClass, "propertyChange",
                                 PropertyChangeListener.class, "propertyChange");

            changed.setDisplayName("Property Change");
            EventSetDescriptor events[] = {changed};
            return events;
      } catch (IntrospectionException e) {
         throw new Error(e.toString());
      }
   }
   public MethodDescriptor[] getMethodDescriptors() {
      try {
         MethodDescriptor showNext
         = new MethodDescriptor(beanClass.getMethod("showNext", null));

            MethodDescriptor methods[] = {showNext};
            return methods;
      } catch (Exception e) {
             throw new Error(e.toString());
      }
   }
   public java.awt.Image getIcon(int iconKind){
      if(iconKind == SimpleBeanInfo.ICON_COLOR_16x16)
         return loadImage("photoAlbumIcon16.gif");
      else
         return loadImage("photoAlbumIcon32.gif");
   }
}
```

La classe viene definita come sottoclasse di `SimpleBeanInfo`, in modo da rendere il processo di sviluppo più rapido.

Il primo metodo, `getPropertyDescriptors`, restituisce un array con un tre `PropertyDescriptor`, uno per ciascuna delle proprietà che si vogliono rendere visibili. Il costruttore di `PropertyDescriptor` richiede quattro argomenti: il nome della proprietà, la classe del Bean, il nome del metodo getter e quello del metodo setter: quest'ultimo è posto a `null`, a significare che le proprietà sono di tipo Read Only. Si noti, in questo metodo e nei successivi, che la creazione dei Descriptors deve essere definita all'interno di un blocco `try-catch`, dal momento che può generare `IntrospectionException`.

Il secondo metodo, `getEventSetDescriptors()`, restituisce un vettore con un unico `EventSetDescriptor`. Quest'ultimo viene inizializzato con quattro parametri: la classe del Bean, il nome della proprietà, la classe dell'ascoltatore e la firma del metodo che riceve l'evento. Si noti la chiamata al metodo `setDisplayName()`, che permette di impostare un nome più leggibile di quello che viene normalmente ottenuto dalle firme dei metodi.

Il terzo metodo, `getMethodDescriptors`, restituisce un vettore contenente un unico `MethodDescriptor`, che descrive il metodo `showNext()`. Il costruttore di `MethodDescriptor` richiede come unico parametro un oggetto di classe `Method`, che in questo esempio viene richiesto alla classe `PhotoAlbum` ricorrendo alla reflection.

Infine il metodo `getIcon()` restituisce un'icona, che normalmente viene associata al Bean all'interno di strumenti visuali.

Per impacchettare il Bean `PhotoAlbum` con le icone e il `BeanInfo`, si può seguire la procedura già descritta, modificando la riga di comando dell'utility `jar` in modo da includere le icone nell'archivio.

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp com\mokabyte\mokabook\
javaBeans\photoAlbum\*.class com\mokabyte\mokabook\javaBeans\
photoAlbum\*.gif com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Personalizzazione dei Bean

L'aspetto e il comportamento di un Bean possono essere personalizzati in fase di composizione all'interno di un tool grafico abilitato ai Beans. Esistono due strumenti per personalizzare un Bean: gli Editor di proprietà e i Customizer. Gli Editor di proprietà sono componenti grafici specializzati nell'editing di un particolare *tipo* di proprietà: interi, stringhe, files… Ogni Editor di proprietà viene associato a un particolare tipo Java, e il tool grafico compone automaticamente un Property Sheet analizzando le proprietà di un Bean, e ricorrendo agli Editor più adatti alla circostanza. In fig. 8.6 si può vedere un esempio di Property Sheet, realizzato dal Bean Box: ogni riga presenta, accanto al nome della proprietà, il relativo Editor.

Un Customizer, d'altra parte, è un pannello di controllo specializzato per un particolare Bean: in questo caso è il programmatore a decidere cosa mostrare nel pannello e in quale

Figura 8.6 – *Un Property Sheet generato in modo automatico a partire dalle proprietà di un pulsante Bean*

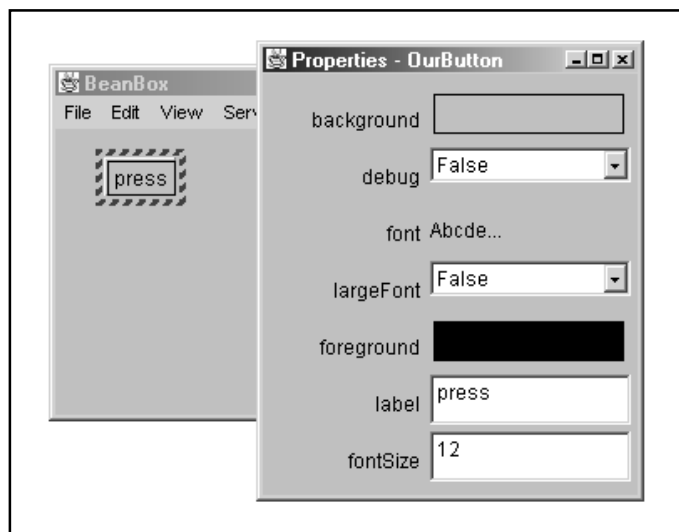
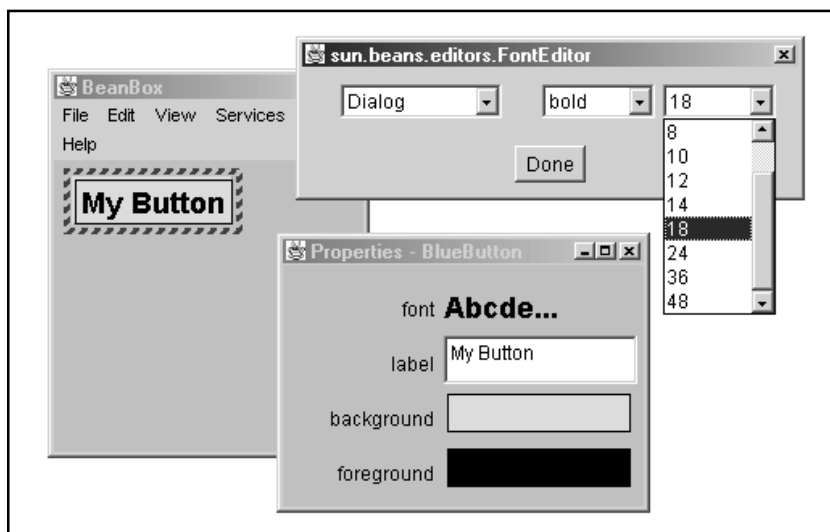



Figura 8.7 – *Il Property Sheet relativo a un pulsante Bean. Si noti il pannello ausiliario* `FontEditor`

maniera. Per questa ragione un Customizer viene associato, grazie al `BeanInfo`, a un particolare Bean e non può, in linea di massima, essere usato su Bean differenti.

Come creare un Editor di proprietà

Un Editor di proprietà deve implementare l'interfaccia `PropertyEditor`, o in alternativa, estendere la classe `PropertyEditorSupport` che fornisce un'implementazione standard dei metodi dell'interfaccia. L'interfaccia `PropertyEditor` dispone di metodi che premettono di specificare come una proprietà debba essere rappresentata in un property sheet. Alcuni Editor consistono in uno strumento direttamente editabile, altri presentano uno strumento a scelta multipla, come un `ComboBox`; altri ancora, per permettere la modifica, aprono un pannello separato, come nella proprietà `font` dell'esempio, che viene modificata grazie al pannello ausiliario `FontEditor`.

Per fornire il supporto a queste modalità di editing, bisogna implementare alcuni metodi di `PropertyEditor`, in modo che ritornino valori non nulli.

I valori numerici o String possono implementare il metodo `setAsText(String s)`, che estrae il valore della proprietà dalla stringa che costituisce il parametro. Questo sistema permette di inserire una proprietà con un normale campo di testo.

Gli Editor standard per le proprietà `Color` e `Font` usano un pannello separato, e ricorrono al Property Sheet solamente per mostrare l'impostazione corrente. Facendo click sul valore, viene aperto l'Editor vero e proprio. Per mostrare il valore corrente della proprietà, è necessario sovrascrivere il metodo `isPaintable()` in modo che restituisca `true`, e sovrascrivere `paintValue` in modo che dipinga la proprietà attuale in un rettangolo all'interno del Property Sheet.

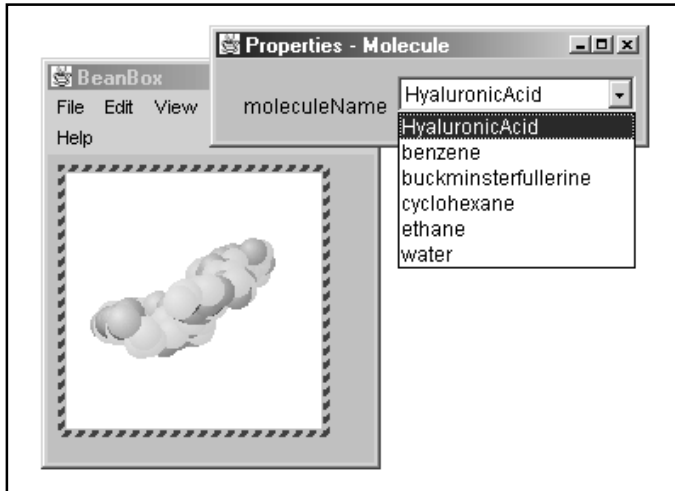
Per supportare l'Editor di Proprietà personalizzato occorre sovrascrivere altri due metodi della classe `PropertyEditorSupport`: `supportsCustomEditor`, che in questo caso deve restituire `true`, e `getCustomEditor`, in modo che restituisca un'istanza dell'Editor.

Registrare gli Editor

I Property Editor vengono associati alle proprietà attraverso un'associazione esplicita, all'interno del metodo `getPropertyDescriptors()` del `BeanInfo`, con una chiamata al metodo `setPropertyEditorClass(Class propertyEditorClass)` del `PropertyDescriptor` corrispondente, come avviene nel Bean `Molecule`

```
PropertyDescriptor pd = new PropertyDescriptor("moleculeName", Molecule.class);
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

In alternativa si può registrare l'Editor con il seguente metodo statico

Figura 8.8 – *Il Bean* `Molecule` *associa alla proprietà* `moleculeName` *di un Editor di proprietà personalizzato*

```
PropertyEditorManager.registerEditor(Class targetType, Class editorType)
```

che richiede come parametri la classe che specifica il tipo e quella che specifica l'Editor.

Customizers

Con un Bean Customizer è possibile fornire un controllo completo sul modo in cui configurare ed editare un Bean. Un Customizer è in pratica una piccola applicazione specializzata nell'editing di un particolare Bean, ogni volta che la configurazione di un Bean richiede modalità troppo sofisticate per il normale processo di creazione automatica del Property Sheet.

Le uniche regole a cui ci si deve attenere per realizzare un Customizer sono:

- deve estendere la classe `Component`, o una delle sue sottoclassi;
- deve implementare l'interfaccia `java.bean.Customizer`;
- deve implementare un costruttore privo di parametri.

Per associare il Customizer al proprio Bean, bisogna sovrascrivere il metodo `getBeanDescriptor` nella classe `BeanInfo`, in modo che restituisca un opportuno

BeanDescriptor, il quale a sua volta dovrà restituire la classe del Customizer alla chiamata del metodo getCustomizerClass.

Serializzazione

Per rendere serializzabile una classe Bean è di norma sufficiente implementare l'interfaccia Serializable, sfruttando così l'Object Serialization di Java. L'interfaccia Serializable non contiene metodi: essa viene usata dal compilatore per marcare le classi che possono essere serializzate. Esistono solo poche regole per implementare classi Serializable: anzitutto è necessario dichiarare un costruttore privo di argomenti, che verrà chiamato quando l'oggetto verrà ricostruito a partire da un file .ser; in secondo luogo una classe serializzabile deve definire al suo interno solamente attributi serializzabili.

Se si desidera fare in modo che un particolare attributo non venga salvato al momento della serializzazione, si può ricorrere al modificatore transient. La serializzazione standard, inoltre, non salva lo stato delle variabili static.

Per tutti i casi in cui la serializzazione standard non risultasse applicabile, occorre procedere all'implementazione dell'interfaccia Externalizable, fornendo, attraverso i metodi readExternal(ObjectInput in) e writeExternal(ObjectOutput out), delle istruzioni esplicite su come salvare lo stato di un oggetto su uno stream e come ripristinarlo in un secondo tempo.

Abstract Windows Toolkit

DI PAOLO AIELLO

Introduzione

Prima di presentare gli strumenti Java che consentono di creare e manipolare interfacce grafiche, è utile richiamare brevemente alla mente le principali caratteristiche delle Graphical User Interfaces (GUI) così come sono implementate praticamente nella totalità degli ambienti grafici esistenti, al di là delle particolarità di ogni ambiente. La principale peculiarità degli ambienti grafici, dal punto di vista del design, rispetto alle interfacce a linea di comando, è che si tratta di ambienti *windowed*, cioè basati sulle finestre, e *event driven*, cioè guidati dagli eventi. Mentre la tipica interazione di un ambiente a linea di comando è sequenziale e basata su un'unica linea di I/O e su un *interprete di comandi* dati in forma di stringhe di testo, la GUI accetta contemporaneamente input diversi provenienti da diversi dispositivi (in genere, tastiera e mouse) e ciascuna ricezione di input da luogo a un *evento* che viene notificato a uno o più *oggetti grafici* incaricati di gestire l'evento, che in risposta eseguono operazioni di vario tipo. Al posto dell'interprete di comandi ci sono delle finestre e al posto delle stringhe di comando stanno gli oggetti grafici con cui l'utente interagisce.

Dal momento che l'ambiente fa un uso intenso della grafica, risulta del tutto naturale che incorpori una serie di servizi grafici standard, che negli ambienti non grafici sono totalmente a carico dei singoli programmi, senza alcun supporto da parte del sistema.

Quindi dall'interfaccia di programmazione (API) di un ambiente grafico ci si possono aspettare tre principali insiemi di servizi: una serie di oggetti grafici concepiti per svolgere compiti di interfaccia utente; un sistema di gestione degli eventi; una serie di funzioni grafiche, che permettono di disegnare sugli oggetti grafici. Dopo questi brevi cenni, anziché procedere con una trattazione teorica della GUI, si passerà direttamente a vedere come questi servizi sono forniti dai package Java, introducendo man mano i vari concetti, con l'aiuto di semplici esempi.

Prima però ecco una breve panoramica su tali package.

Il package AWT

Il package `java.awt` (AWT sta per Abstract Windows Toolkit) è il primo framework Java concepito per gestire le GUI. Soprattutto la prima versione (`1.0`) era caratterizzata da un design particolarmente poco riuscito che gli aveva fruttato una fama tutt'altro che lusinghiera. Probabilmente — dati gli obiettivi molto più ristretti, rispetto agli attuali, del "progetto Java" — inizialmente non si era dedicata la dovuta attenzione a questo aspetto della programmazione. I difetti principali di `AWT 1.0` si possono riassumere come segue: una penuria di oggetti grafici e di funzionalità, dovuta al fatto che il toolkit, a causa della natura multipiattaforma di Java, doveva poter funzionare in qualunque ambiente e per garantire questa portabilità si era creato un toolkit che comprendeva solo oggetti e funzionalità comuni a tutte le piattaforme; la non perfetta portabilità dovuta al fatto che, essendo usati oggetti nativi dell'ambiente "ospite", questi sono in realtà differenti nell'aspetto e in piccoli particolari da ambiente ad ambiente, quindi il *look and feel* di un'applicazione varia a seconda della piattaforma; un sistema di gestione degli eventi decisamente mal concepito sia dal punto di vista del design Object Oriented che della funzionalità.

Quest'ultimo difetto fu corretto con `AWT 1.1`, che introdusse un sistema alternativo di gestione degli eventi, molto più funzionale e, dal punto di vista del design, probabilmente il migliore e più avanzato tra quelli degli attuali framework grafici di qualsiasi linguaggio. Anche le funzionalità vennero estese e migliorate, ma senza risolvere il problema alla radice.

Swing e JFC

Restavano gli altri due difetti, superati solo in Java 2 con il package `javax.swing`. Nonostante il package faccia parte delle "estensioni", contraddistinte dal prefisso `javax`, si tratta di un package di fondamentale importanza, destinato a soppiantare del tutto in futuro il vecchio `AWT`. Le Swing API fanno parte di un insieme di packages, conosciuto come *Java Foundation Classes* (JFC) che comprende anche una serie di altre librerie, sempre dedicate alla gestione delle interfacce e della grafica.

Con `swing` vengono riscritti tutti gli oggetti grafici, basandosi su una concezione del tutto diversa dalla precedente: gli oggetti utilizzati non sono più quelli nativi del sistema, ma degli oggetti Java creati a partire dalle funzioni di basso livello del sistema. In tal modo si possono usare esattamente gli stessi oggetti con lo stesso identico aspetto e le stesse funzionalità in sistemi diversi, e soprattutto non sussiste più la limitazione dell' "insieme minimo" di oggetti e funzioni comuni a tutti gli ambienti.

Nelle intenzioni degli implementatori, il package `Swing` dovrebbe sostituire e rendere obsoleto il package `AWT`, pur conservando l'eredità di alcune classi di base. Tuttavia que-

sto sarà veramente possibile solo quando `Swing` sarà supportato dalla gran maggioranza dei browser in circolazione. Fino a quel momento, almeno per le applet, `AWT` rimane una scelta pressoché obbligata, nonostante le sue evidenti limitazioni.

Le Applet

Un discorso a parte merita il package `java.applet`. Anche se in sé esso non introduce elementi di interfaccia veri e propri, ma solo le funzioni necessarie a integrare oggetti grafici all'interno di un browser, le applet fanno parte a tutti gli effetti delle classi GUI, essendo derivate da una classe AWT. Inutile sottolineare l'importanza delle applets nella storia e nella diffusione di Java. Per quanto si possano senza dubbio annoverare tra i punti di forza di Java, le Applet soffrono delle limitazioni dovute alla dipendenza dai browser, con le loro implementazioni sempre vecchie rispetto alle versioni correnti del linguaggio, della Virtual Machine. Ora come ora, le Applet sono l'unica ragione per cui sopravvivono le vecchie classi AWT e il principale ostacolo alla diffusione degli oggetti Swing, utilizzabili per ora sono in applicazioni stand-alone o in ambito intranet in cui sia possibile installare senza problemi le librerie mancanti o il java-plugin, come si vedrà nel capitolo sulle Applet.

Questi tre argomenti saranno affrontati in capitoli separati. Il presente capitolo è dedicato interamente al package AWT.

Una panoramica su AWT

In questa sezione si affronteranno il modo in cui è stato concepito il package `AWT` e le principali funzionalità che offre. A tal fine si farà riferimento all'esempio riportato integralmente qui sotto.

```
import java.awt.*;
import java.awt.event.*;

public class HelloGoodbyFrame extends Frame {
   // classi interne per la gestione degli eventi

   class ButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
         Button source = (Button)event.getSource();
         String message = null;
         if (source == helloButton)
            sayHello();
         else if (source == goodbyButton)
            sayGoodby();
      }
   }

   class FrameListener extends WindowAdapter {
```

```
      public void windowClosed(WindowEvent e) {
         System.exit(0);
      }
   }

   // layout
   BorderLayout layout = new BorderLayout();
   // componenti
   Panel contentPanel = new BorderedPanel();
   Button helloButton = new Button();
   Button goodbyButton = new Button();
   Label messageLabel = new Label();
   // listener
   FrameListener frameListener = new FrameListener();
   ButtonListener buttonListener = new ButtonListener();

   public HelloGoodbyFrame() {
      // imposta le proprietà del Frame
      this.setTitle("primo esempio AWT");
      this.setSize(500, 60);

      // imposta il layout del Frame
      this.setLayout(layout);

      // imposta le proprietà dei pulsanti
      helloButton.setLabel("Presentati");
      goodbyButton.setLabel("Congedati");

      // inserisce i pulsanti e la label nel Panel
      contentPanel.add(helloButton);
      contentPanel.add(goodbyButton);
      contentPanel.add(messageLabel);

      // inserisce il Panel nel Frame
      this.add(contentPanel, BorderLayout.CENTER);

      // assegna il listener al frame
      this.addWindowListener(frameListener);

      // collega il listener ai pulsanti
      helloButton.addActionListener(buttonListener);
      goodbyButton.addActionListener(buttonListener);
   }

   public void sayHello() {
      messageLabel.setText("Ciao! Sono il primo esempio AWT");
      validate();
   }

   public void sayGoodby() {
```

```
      messageLabel.setText("Ora devo andare. Arrivederci!");
      validate();
      try {
         Thread.sleep(2000);
      }
      catch (InterruptedException e) {}
      dispose();
   }

   public static void main(String[] args) {
      new HelloGoodbyFrame().show();
   }
}


class BorderedPanel extends Panel {
   BorderedPanel() {
      FlowLayout layout = (FlowLayout)getLayout();
      layout.setAlignment(FlowLayout.LEFT);
   }

   public void paint(Graphics g) {
      super.paint(g);
      Dimension size = getSize();
      g.setColor(Color.red);
      g.drawRect(3, 3, size.width -6, size.height -6);
   }
}
```

Componenti: contenitori e controlli

In AWT tutti gli oggetti grafici di interfaccia sono chiamati *componenti* e derivano da un'unica classe astratta chiamata `Component`. I componenti si dividono in due categorie principali: i *contenitori*, derivati dalla classe astratta `Container` (sottoclasse di `Component`), concepiti per fare appunto da contenitori per altri componenti; e i *controlli*, che forniscono tutte le funzionalità necessarie per relizzare un'interfaccia grafica.

Ai contenitori sono associati degli oggetti particolari detti `layout`, che determinano il modo in cui i controlli vengono posizionati all'interno del contenitore

Parlando in generale, ogni componente è caratterizzato da una serie di *proprietà* su alcune delle quali il programmatore può intervenire modificandole in qualche misura, e da un certo *comportamento*, dipendente dalle sue funzionalità specifiche, che riguarda principalmente il suo modo di interagire con l'utente e con gli altri controlli, e che si concretizza con le modalità con cui gestisce gli eventi (a quali eventi risponde e come reagisce a questi eventi). Nell'esempio `HelloGoodbyFrame.java`, abbiamo un contenitore di tipo `HelloGoodbyFrame`, derivato dalla classe `Frame`, a sua volta derivata da

`Window`, che è il capostipite di tutti gli oggetti di tipo window, cioè quelli che fungono da "contenitori principali" (*top-level window*) ossia non a loro volta contenuti da altri oggetti. Si precisa che questa nomenclatura, che limita la categoria window alle sole top-level window, vale solo per AWT; in altri contesti una window può indicare un qualunque oggetto grafico, che sia o meno top-level, e che sia o meno un contenitore.

L'esempio mostra il modo in cui viene definita una `Window`: in osservanza delle regole della programmazione a oggetti essa racchiude tutto il codice necessario per il suo funzionamento e tutti gli oggetti che compaiono al suo interno.

Per il momento si esamina la parte di codice che si riferisce alla creazione e manipolazione degli oggetti, qui sotto riportata estrapolata dall'insieme.

```
public class HelloGoodbyFrame extends Frame {
   // componenti
   Panel contentPanel = new BorderedPanel();
   Button helloButton = new Button();
   Button goodbyButton = new Button();
   Label messageLabel = new Label();

   public HelloGoodbyFrame() {
      // imposta le proprietà del Frame
      this.setTitle("primo esempio AWT");

      // imposta le proprietà dei pulsanti
      helloButton.setLabel("Presentati");
      goodbyButton.setLabel("Congedati");

      // inserisce i pulsanti e la label nel Panel
      contentPanel.add(helloButton);
      contentPanel.add(goodbyButton);
      contentPanel.add(messageLabel);

      // inserisce il Panel nel Frame
      this.add(contentPanel, BorderLayout.CENTER);
   }
}
```

Per prima cosa vengono definite delle variabili della classe che contengono i riferimenti ai *componenti* che verranno visualizzati all'interno del `Frame`:

- Un pannello (*Panel*) `contentPanel`. Un pannello è un contenitore, ma non una `Window`, quindi deve stare all'interno di un altro contenitore. Anzichè inserire direttamente i controlli nel `Frame`, si crea un pannello (di tipo `BorderedPanel`, sottoclasse di `Panel`, che si vedrà in seguito) che verrà inserito nel `Frame`, e si mettono i controlli nel `Panel`. I motivi di ciò si vedranno in seguito, quando si parlerà della grafica.

- Due pulsanti (*Button*) `helloButton` e `goodbyButton`. I pulsanti sono dei semplici controlli che compiono una determinata operazione in risposta a un click del mouse.

- Una etichetta (*Label*) `messageLabel`. Le etichette sono degli oggetti che visualizzano del testo, solitamente senza interagire con l'utente (cioè senza rispondere a eventi generati direttamente dall'utente).

Il Frame e i componenti vengono inizializzati nel costruttore della classe. Per prima cosa vengono definite alcune proprietà del Frame: il *titolo*, ossia la scritta che comparirà nella barra del titolo, e le dimensioni. Quindi vengono impostate le proprietà dei pulsanti; in questo caso ci si limita ad assegnare il testo visualizzato sul pulsante, con il metodo `Button.setLabel()`. A questo punto i componenti vengono inseriti nel `Panel`, e il `Panel` nel `Frame`, utilizzando il metodo `Container.add()`.

I Layout

Nella maggior parte dei framework grafici che vengono normalmente utilizzati con altri linguaggi, soprattutto in ambiente Windows, la posizione dei controlli all'interno dei contenitori è determinata secondo un sistema di coordinate relativo, la cui origine è solitamente il vertice superiore sinistro del contenitore. In AWT, anche se è possibile utilizzare, volendo, questa modalità, generalmente il posizionamento dei componenti viene affidato ai `layout`, oggetti a sé stanti che possono essere associati a uno o più contenitori. Una volta assegnato al contenitore, il layout assegnerà, ai componenti inseriti all'interno del contenitore, delle posizioni calcolate secondo determinati algoritmi, a mano a mano che i componenti vengono inseriti. Ci sono vari tipi di layout, che più avanti saranno esaminati in dettaglio. I contenitori hanno in genere un layout di default, che utilizzano se nessun oggetto `Layout` viene loro assegnato esplicitamente.

Nell'esempio vengono usati due tipi di `Layout`: il primo è il `FlowLayout`, che dispone i componenti uno dopo l'altro nell'ordine in cui vengono inseriti (da sinistra a destra, poi eventualmente "andando a capo" come nelle righe di un documento di testo, con i componenti che possono essere allineati a sinistra, a destra, al centro); il secondo è il `BorderLayout`, che dispone i componenti in aree predefinite (centro, nord, est, sud, ovest) corrispondenti al centro e ai lati del contenitore, con i componenti che vengono ridimensionati in modo da occupare tutta l'area loro riservata.

Ecco il codice relativo ai layout nella classe `HelloGoodbyFrame`:

```
public class HelloGoodbyFrame extends Frame {
   // layout
   BorderLayout layout = new BorderLayout();
   // componenti
   Panel contentPanel = new BorderedPanel();
   Button helloButton = new Button();
   Button goodbyButton = new Button();
   Label messageLabel = new Label();
```

```
   public HelloGoodbyFrame() {
      // imposta le proprietà del Frame
      this.setTitle("primo esempio AWT");

      // imposta il layout del Frame
      this.setLayout(layout);

      // imposta le proprietà dei pulsanti
      helloButton.setLabel("Presentati");
      goodbyButton.setLabel("Congedati");

      // inserisce i pulsanti e la label nel Panel
      contentPanel.add(helloButton);
      contentPanel.add(goodbyButton);
      contentPanel.add(messageLabel);

      // inserisce il Panel nel Frame
      this.add(contentPanel, BorderLayout.CENTER);
   }
}
```

Qui è stato semplicemente creato un oggetto di tipo `BorderLayout`, e nel costruttore è stato assegnato al `Frame`, usando il metodo `setLayout()`. In realtà è stata compiuta un'operazione superflua, perché un `Frame` ha un layout di default che è appunto un `BorderLayout`; il codice è giustificato solo dai fini esemplificativi. Si noti che nel metodo `add()` usato per inserire il pannello nel `Frame`, c'è un secondo argomento che specifica un *constraint*, cioè una modalità particolare con cui il layout posiziona il componente. In questo caso si tratta di una costante `BorderLayout.CENTER`, che indica la posizione in cui va inserito il componente. Anche qui non era necessario specificare, dato che il centro è la posizione di default per il `BorderLayout`.

Ecco ora la classe `BorderedPanel`.

```
class BorderedPanel extends Panel {
   BorderedPanel() {
      FlowLayout layout = (FlowLayout)getLayout();
      layout.setAlignment(FlowLayout.LEFT);
   }
}
```

In questo caso si è usato il layout di default, che nel caso del `Panel` è un `FlowLayout`, che viene restituito dal metodo `getLayout()`; poiché l'allineamento di default del `FlowLayout` è al centro mentre qui serve a sinistra, viene modificato con il metodo `setAlignment()`.

Abbiamo quindi un `Frame` con un `BorderLayout`, nel quale inseriamo il `Panel`. Questo viene inserito al centro e, poiché i lati sono liberi (visto che è stato inserito nien-

t'altro che il pannello), il pannello occupa l'intera area del `Frame`. Il pannello, a sua volta, ha un `FlowLayout`, quindi i componenti vengono inseriti in fila uno dopo l'altro. Notare che sono state assegnate al `Frame` delle dimensioni tali da consentire una visualizzazione ottimale dei componenti con il `FlowLayout`.

La gestione degli eventi

Finora è stata analizzata la parte statica dell'esempio. Qui si descrive la parte dinamica, ossia il comportamento dei vari componenti in risposta agli eventi generati dall'utente, da altri componenti, o dal sistema.

A partire da AWT 1.1, gli eventi vengono gestiti per mezzo di oggetti *listener* (ascoltatori) che implementano determinate interfacce. Ogni tipo di listener resta "in ascolto" di un determinato evento, o di un certo insieme di eventi. Per ogni tipo di evento deve essere implementato un metodo che "gestisce" l'evento, cioè che viene chiamato quando l'evento si verifica. Un listener può essere assegnato a uno o più componenti, che a quel punto saranno sotto il suo controllo, e ogni volta che un evento di un tipo gestito dal listener sara generato *per uno dei componenti tenuti sotto controllo*, verrà eseguito il metodo corrispondente.

Potendo liberamente assegnare i listener ai componenti (purché compatibili con la natura del componente stesso) è possibile sia che un listener controlli più componenti, sia che un componente sia controllato da più listener.

Questo è il codice relativo alla gestione degli eventi.

```
public class HelloGoodbyFrame extends Frame {
   // classi interne per la gestione degli eventi

   class ButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
         Button source = (Button)event.getSource();
         String message = null;
         if (source == helloButton)
            sayHello();
         else if (source == goodbyButton)
            sayGoodby();
      }
   }

   class FrameListener extends WindowAdapter {
      public void windowClosed(WindowEvent e) {
         System.exit(0);
      }
   }

   // layout
```

```
BorderLayout layout = new BorderLayout();
// componenti
Panel contentPanel = new BorderedPanel();
Button helloButton = new Button();
Button goodbyButton = new Button();
Label messageLabel = new Label();
// listener
FrameListener frameListener = new FrameListener();
ButtonListener buttonListener = new ButtonListener();

public HelloGoodbyFrame() {
   // imposta le proprietà del Frame
   this.setTitle("primo esempio AWT");
   this.setSize(500, 60);

   // imposta il layout del Frame
   this.setLayout(layout);

   // imposta le proprietà dei pulsanti
   helloButton.setLabel("Presentati");
   goodbyButton.setLabel("Congedati");

   // inserisce i pulsanti e la label nel Panel
   contentPanel.add(helloButton);
   contentPanel.add(goodbyButton);
   contentPanel.add(messageLabel);

   // inserisce il Panel nel Frame
   this.add(contentPanel, BorderLayout.CENTER);

   // assegna il FrameListener al frame
   this.addWindowListener(frameListener);

   // collega il ButtonListener ai pulsanti
   helloButton.addActionListener(buttonListener);
   goodbyButton.addActionListener(buttonListener);
}

public void sayHello() {
   messageLabel.setText("Ciao! Sono il primo esempio AWT");
   validate();
}

public void sayGoodby() {
   messageLabel.setText("Ora devo andare. Arrivederci!");
   validate();
   try {
      Thread.sleep(2000);
   }
```

```
      catch (InterruptedException e) {}
      dispose();
   }
}
```

Per prima cosa, sono state create delle inner class per la gestione degli eventi. La scelta va a favore delle inner class anziché di classi esterne per esemplificare una implementazione molto diffusa dei listener, per i quali può risultare utile, se non indispensabile, poter accedere ai dati della classe "contenitore" (*outer class*), anche a quelli privati. Si ricorda che con le inner class questo è possibile senza dover passare all'oggetto inner class il reference dell'oggetto outer class; ossia un metodo della inner class ha, nei confronti dell'outer class, le stesse proprietà dei metodi dell'outer class stessa. In realtà non sempre è conveniente implementare i listener come inner class. Nel seguito questo aspetto sarà discusso in maniera più approfondita.

La prima di queste classi interne, che è stata chiamata `ButtonListener`, implementa l'interfaccia `ActionListener`. Un `ActionListener` è un listener che risponde a un `ActionEvent` che si può considerare come l' "evento principale" di un componente. Più precisamente si tratta di un evento specifico per ogni componente, generato a partire da eventi differenti, a seconda della natura del componente. Per un `Button` un `ActionEvent` viene generato quando il pulsante viene premuto, in genere con un click del mouse (che genera un `MouseEvent`) ma anche eventualmente con un input da tastiera (`KeyEvent`).

L'interfaccia `ActionListener` comprende un unico metodo `actionPerformed`, che viene implementato dalla nostra classe, e che riceve come parametro un oggetto di tipo `ActionEvent` (sottoclasse di `AWTEvent`). Questo oggetto contiene diverse informazioni sul contesto che ha generato l'evento, tra cui l'oggetto a cui l'evento si riferisce, restituito dal metodo `getSource()`. Avendo ottenuto il reference a questo oggetto, esso può essere confrontato con quelli dei `Button` presenti nel `Frame` per individuare la sua esatta identità ed eseguire il codice appropriato. In questo caso si chiama, a seconda del pulsante premuto, uno dei due metodi `sayHello()` e `sayGoodby()`.

La seconda classe, `FrameListener`, serve per gestire l'evento di chiusura del `Frame`. A tal fine deve essere utilizzato un oggetto di tipo `WindowListener`. Ma l'interfaccia `WindowListener`, a differenza di `ActionListener`, non si limita a gestire un solo tipo di evento, e quindi contiene diversi metodi. Dato che in questo caso interessa gestirne uno soltanto, per evitare di implementare inutilmente tutti gli altri metodi, lasciandoli vuoti, anzichè implementare l'interfaccia, si estende la classe `WindowAdapter`, che implementa tutti metodi con corpo vuoto. In tal modo basta ridefinire solo il metodo che interessa, cioè `windowClosed()`, che gestisce l'evento omonimo. In questo metodo viene chiamato `System.exit()` per terminare l'applicazione quando il `Frame` viene chiuso.

Dopo aver definito le classi, si crea un'istanza di ciascuna come variabile della classe `Frame`, e nel costruttore si assegna il listener ai componenti con i metodi `Window.addWindowListener()` e `Button.addActionListener()`.

Infine, si definiscono i metodi `sayHello()` e `sayGoodby()`, che vengono chiamati dal `ButtonListener`. `sayHello()` imposta il testo visualizzato dalla label, inizialmente vuoto, tramite il metodo `setText()`. `sayGoodby()`, dopo aver modificato il testo della `Label`, resta in attesa per 2 secondi, quindi chiude il `Form` con il metodo `dispose()`.

Da notare, in entrambi i casi, l'uso del metodo `Container.validate()`, che deve essere chiamato dopo ogni modifica che possa provocare un riposizionamento o ridimensionamento dei componenti da parte del layout. In questo caso il layout ridimensiona la label a seconda del testo contenuto, quindi se non si chiama `validate()`, le dimensioni del controllo rimarranno quelle iniziali, con una lunghezza prossima allo `0`, che è quella della `Label` vuota, anche dopo la modifica del testo.

La grafica

È arrivato il momento di esaminare un metodo della classe `BorderedPanel`, che finora è stato lasciato da parte. Si tratta del metodo `paint()`, ereditato dalla classe `Component`. Questo è un metodo che viene chiamato ogni volta che l'oggetto, per qualunque motivo, necessita di essere disegnato o ridisegnato. In sostanza è quello che determina come il componente viene visualizzato.

```
class BorderedPanel extends Panel {
   BorderedPanel() {
      FlowLayout layout = (FlowLayout)getLayout();
      layout.setAlignment(FlowLayout.LEFT);
   }

   public void paint(Graphics g) {
      super.paint(g);
      Dimension size = getSize();
      g.setColor(Color.red);
      g.drawRect(3, 3, size.width -6, size.height -6);
   }
}
```

Il metodo `paint()` prende come parametro un oggetto `Graphics`, che in AWT è lo strumento adoperato per qualunque operazione di disegno e visualizzazione (figure geometriche, testo, immagini). Usando i metodi di quest'oggetto si può disegnare direttamente sulla superficie del componente. In questa maniera si possono, in sostanza, creare nuovi tipi di componenti.

Nell'esempio è stato ridefinito il metodo `paint()`, e usato l'oggetto `Graphics` per disegnare una cornice di colore rosso a 3 pixel di distanza dal bordo del pannello.

Il metodo paint() sarebbe in teoria ridefinibile anche direttamente nella classe HelloGoodbyFrame. In pratica, almeno in alcune Virtual Machine in ambiente MS-Windows, anche se il metodo paint() viene regolarmente eseguito, non viene visualizzato nulla. Questo accade per le classi Frame e Dialog e le loro sottoclassi. Di conseguenza risulta necessario ricorrere a pannelli inseriti nelle finestre principali. Si ricordi che in AWT i componenti sono oggetti dell'ambiente grafico sottostante (vedere più avanti "Componenti AWT e peer"), per cui il funzionamento è condizionato da particolari implementativi di basso livello della VM, relativi alla particolare piattaforma, e dall'implementazione dei componenti nativi stessi, che sfuggono al controllo del programmatore Java.

Esecuzione dell'applicazione

Come si può vedere dal codice dal main() mandare in esecuzione l'applicazione AWT è estremamante semplice.

```
public static void main(String[] args) {
   new HelloGoodbyFrame().show();
}
```

Basta creare un oggetto HelloGoodbyFrame e chiamare il suo metodo show() che rende visibile l'oggetto. Tutto il resto, compresa la terminazione dell'applicazione, è gestito dalla classe stessa.

Riepilogo

Si riassumono i vari passi con cui è stata costruita la nostra piccola applicazione, composta dalle due classi HelloGoodbyFrame e BorderedPanel.

- Per la classe BorderedPanel: definire una classe BorderedPanel, sottoclasse di Panel; modificare l'allineamento del layout del BorderedPanel, nel costruttore; ridefinire il metodo paint() del BorderedPanel, per disegnare una cornice sul pannello.

- Per la classe HelloGoodbyFrame: definire una classe di tipo Window, in questo caso una sottoclasse di Frame; creatre un oggetto layout assegnandolo a una variabile della classe; creare un BorderedPanel e dei controlli assegnandoli a variabili della classe; definire una inner class ButtonListener per gestire gli eventi degli

oggetti `Button`, con un metodo `ActionPerformed()` in cui inserire il codice da eseguire in risposta all'evento; definire un'altra inner class per gestire l'evento `WINDOW_CLOSED` del `Frame`, in modo che alla chiusura del `Frame` l'applicazione termini; definire i metodi `sayHello()` e `sayGoodby()` utilizzati dal `ButtonActionListener`.

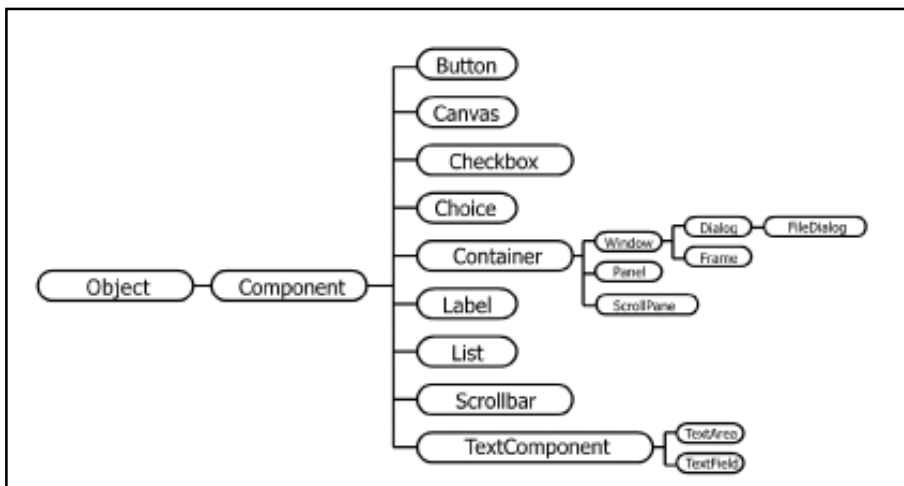
- Nel costruttore: impostare alcune proprietà del `Frame` e dei controlli; assegnare il `layout` al `Frame`; inserire i controlli all'interno del `Panel`, e il `Panel` all'interno del `Frame`; assegnare i rispettivi listener al `Frame` e a i pulsanti.

- Nel `main()`: creare un oggetto `HelloGoodbyFrame` e chiamare il metodo `show()` per visualizzarlo.

I componenti

Componenti AWT e Peer

Prima di esaminare in dettaglio i singoli componenti, descriviamo brevemente il modo in cui la virtual machine implementa questi componenti. Si è già accennato al fatto che in AWT i componenti sono in effetti componenti *nativi* dell'ambiente grafico sottostante, e quindi oggetti differenti a seconda della piattaforma. I componenti AWT sono quindi in realtà solo delle interfacce verso questi componenti nativi, detti *peer* (pari, simile) per indicare che sono una sorta di *alter ego* del componente nel sistema sottostante. Le funzio-

Figura 9.1 – *La gerarchia delle classi derivate da* `java.awt.Component`

nalità degli oggetti non sono quindi implementate in Java, ma direttamente nel codice del sistema, per cui il comportamento del componente Java è subordinato a quanto offerto dal sistema operativo. Inoltre, la necessità di poter ususfruire delle stesse funzionalità indipendentemente dalla piattaforma costringe a limitare notevolmente il numero di componenti e le loro funzionalità. Un'altra conseguenza dell'uso dei peer è che le applicazioni Java realizzate con AWT avranno un *look and feel* differente in ogni piattaforma.

La gerarchia `Component`

In fig. 9.1 è mostrata l'intera gerarchia di classi derivate da `java.awt.Component`. Come si vede, si tratta di un insieme piuttosto limitato, che comprende solo i componenti comuni a tutti gli ambienti grafici delle diverse piattaforme supportate da Java.

La classe `Component`, dal canto suo, fornisce una numerosa serie di servizi di base utilizzati da tutte le sottoclassi. Lasciando da parte tutte le funzionalità legate ai layout, alla gestione degli eventi e alla grafica, che saranno affrontate nelle prossime sezioni, si descrivono brevemente le altre funzionalità dedicate principalmente al controllo delle dimensioni e della posizione e di alcuni attributi generali.

Identità

Ogni componente è identificato da un nome, assegnato a runtime (da non confondersi con il nome della variabile che lo rappresenta), che può essere usato per identificare il componente in alternativa al valore del suo reference. Inoltre ogni componente, per essere visualizzato, deve stare all'interno di un altro componente, il suo parent, a meno che non si tratti di una top-level window (si veda la sezione "Container").

```
Container getParent()
```

Restituisce l'oggetto entro il quale il componente è contenuto; se il valore è *null*, il componente sta direttamente sullo schermo.

```
void setName(String s)
String getName()
```

Assegnano e restituiscono il nome del componente.

Posizione e dimensioni

I seguenti metodi possono essere utilizzati per determinare le dimensioni e la posizione di un componente sullo schermo o all'interno del suo contenitore. In genere questi meto-

di hanno effetto solo in assenza di layout. Infatti questi, se presenti, determinano dimensioni e posizione dei componenti in base al loro specifico algoritmo.

Per questi parametri spaziali sono utilizzate alcune classi di supporto: `Point`, che rappresenta una coppia di coordinate `x y`; `Dimension`, che rappresenta una dimensione formata da `width` e `height`; `Rectangle`, che si può considerare una combinazione dei precedenti, che rappresenta un rettangolo di posizione e dimensioni determinate.

```
void setLocation(Point p)
void setLocation(int x, int y)
```

Impostano la posizione relativa al container

```
Point getLocation()
void getLocation(Point p)
```

Restituiscono la posizione relativa al container; nella seconda forma viene passato un parametro di output di tipo `Point`.

```
getLocationOnScreen()
```

Restituisce la posizione relativa all'intero schermo.

```
getX()
getY()
```

Restituiscono le singole coordinate.

```
void setSize(Dimension d)
void setSize(int width, int height)
```

Assegnano la dimensione.

```
Dimension getSize()
void getSize(Dimension d)
```

Restituiscono la dimensione, la seconda forma con un parametro di output.

```
getWidth()
getHeight()
```

Restituiscono singolarmente larghezza (dimensione x) e altezza (dimensione y).

```
void setBounds(Rectangle r)
void setBounds(int x, int y, int width, int height)
```

Impostano posizione relativa al container e dimensioni.

```
Rectangle getBounds()
void getBounds(Rectangle r)
```

Restituiscono un oggetto `Rectangle` che rappresenta le dimensioni e la posizione relativa al container del componente.

```
boolean contains(int x, int y)
boolean contains(Point p)
```

Indicano se il punto specificato si trova all'interno dell'area del componente.
Ci sono altri metodi relativi alla posizione e alle dimensioni del componente che vengono usati con i layout. Saranno trattati nella sezione *Layout*.

Visibilità

```
boolean isDisplayable()
```

Indica se il componente è visualizzabile (solo dal JDK 1.2 in poi). Un componente è visualizzabile se è una top-level window oppure se è inserito all'interno di un container a sua volta visualizzabile.

```
void setVisible(boolean v)
boolean isVisible()
```

Impostano e indicano se il componente è (virtualmente) visibile. Questa proprietà impostata a `true` non determina necessariamente la visualizzazione del componente.

```
boolean isShowing()
```

Restituisce `true` solo se il componente è effettivamente visualizzato, ossia se è displayable, è visible, e il suo parent è a sua volta displayable e visible.

Focus

Il focus è la proprietà che determina qual è il "componente corrente", o focalizzato. Il componente con il focus è quello che per primo riceve l'input da tastiera, a volte viene visualizzato in maniera particolare e, se si tratta di una finestra, sta in genere in primo piano, anche se la proprietà che determina il "piano" di visualizzazione (z-order) non è necessariamente correlata con il focus. Il focus può essere ottenuto solo da un componente attualmente visualizzato e attivo (enabled). Generalmente il focus viene assegnato dall'utente con un click del mouse sul componente prescelto, ma può anche esser assegnato da codice.

```
boolean hasFocus()
```

Indica se il componente detiene il focus.

```
requestFocus()
```

Richiede per il componente il focus; questo sarà trasferito solo se il componente è visualizzato e attivo.

```
boolean isFocusTraversable()
```

Indica se il componente fa parte della lista di componenti che ricevono il focus da tastiera con il tasto `TAB`. Se il metodo restituisce `false` il focus può essere ugualmente ottenuto con `requestFocus()`.

```
void transferFocus()
```

Trasferisce il focus al prossimo componente (visualizzato e attivo) incluso nel contenitore.

Altre proprietà

```
void setCursor(Cursor c)
```

```
Cursor getCursor()
```

Assegna e restituisce il tipo di cursore del mouse. A ogni tipo di cursore corrisponde un'icona specifica. La classe `Cursor` definisce una serie di cursori predefiniti, e in più offre la possibilità di aggiungere cursori definiti dall'utente. Quando il puntatore del mouse entra nell'area del componente, viene visualizzato con l'icona corrispondente al cursore impostato.

```
void setEnabled(boolean b)
boolean isEnabled()
```

Riguardano lo stato di attività o meno del componente; se un componente è inattivo, non riceve input da tastiera o da mouse; generalmente viene visualizzato in maniera particolare, per indicare il suo stato.

Si vedrà ora la descrizione di ciascun componente, cominciando dai `Container`.

Container

I Container si suddividono in due categorie: le *top-level window*, cioè le finestre di primo livello, che non stanno all'interno di altre finestre (ma direttamente sull'area dello schermo o desktop, come a volte viene chiamata) e i *panel*, che per essere visualizzati devono essere inseriti all'interno di altri Containers.

In una interfaccia grafica, normalmente, i componenti sono contenuti uno dentro l'altro, formando una vera e propria *gerarchia* di contenimento, con una struttura ad albero. Naturalmente questa gerarchia non ha nulla a che fare con la gerarchia di classi entro la quale sono organizzati i componenti. In Java ci si riferisce al componente contenitore con il termine *parent* e un componente contenuto con il termine *child.*

La classe `Container`, capostipite di tutti i contenitori, fornisce una serie di metodi per la gestione dei componenti inclusi all'interno del contenitore, i cui reference sono mantenuti in un array.

```
void add(Component c)
void add(Component c, int index)
```

Inseriscono un componente nel container, in coda alla lista oppure in una specifica posizione. Ci sono altri metodi `add()` utilizzati con i layout, che saranno trattati nella sezione *Layout*.

```
void remove(Component c)
```

```
void remove(int index)
void removeAll()
```

Rimuovono dal contenitore, rispettivamente, il dato componente, il componente di dato indice, tutti i componenti.

```
int getComponentCount()
```

Restituisce il numero di componenti inclusi nel Container.

```
Component[] getComponents
```

Restituisce l'array con tutti i compïnenti del Container.

```
Component getComponent(int index)
```

Restituisce il componente di dato indice.

```
Component getComponentAt(int x, int y)
Component getComponentAt(Point p)
```

Restituiscono il componente entro il quale si trova il punto specificato; se nessun componente "figlio" contiene il punto; viene restituito un reference al Container stesso, se questo contiene il punto, ovvero `null` se il punto si trova all'esterno del Container.

```
Component findComponentAt(int x, int y)
Component findComponentAt(Point p)
```

Funzionano in modo simile a `getComponentAt()`. La differenza è che, se trovano un componente "figlio" che è a sua volta un Container, eseguono la ricerca in maniera ricorsiva all'interno di questo, fino ad arrivare a un componente che non contiene a sua volta altri componenti.

```
boolean isAncestorOf(Component c)
```

Restituisce `true` se il componente specificato è contenuto (direttamente o indirettamente) all'interno del Container.

Top-level window

La classe `Window` è la capostipite delle top-level window, dal punto di vista della gerarchia delle classi. Se però si considera il ruolo nell'ambito di un'applicazione, si può considerare la classe `Frame` come la classe al vertice della gerarchia. Infatti gli oggetti `Frame` sono gli unici veramente indipendenti, nel senso che non devono avere un *owner*, cioè non devono dipendere da un'altra `Window`.

Si prenderà quindi in considerazione prima la classe `Window` per i metodi che introduce come classe base, mentre si descriverà il funzionamento delle finestre, anche con esempi, a partire dalla classe `Frame`.

La classe `Window`

La classe `Window`, pur essendo una classe non astratta, non viene generalmente istanziata direttamente, e si preferisce usare le sottoclassi `Frame` e `Dialog`, che presentano funzionalità più complete. Nella classe `Window` sono presenti i seguenti metodi relativi alla gestione generale delle finestre.

```
Window(Window owner)
Window(Frame owner)
```

L'owner deve essere specificato nel costruttore e non può essere `null`.

```
Window getOwner()
```

Restituisce la Window "di appartenenza". Il concetto di owner è diverso da quello di parent, introdotto prima. Una Window, anche se top-level, ossia non contenuta in nessun altra Window, può "appartenere", ossia essere dipendente, da un'altra Window. Questo implica, tra l'altro, che la chiusura della owner window determina anche la chiusura delle window dipendenti.

```
Window[] getOwnedWindows
```

Restituisce un array che contiene tutte le Window dipendenti dalla Window di cui viene chiamato il metodo (solo dal JDK 1.2 in poi).

```
void toFront()
```

```
void toBack()
```

Mettono la finestra rispettivamente in primo piano e sullo sfondo, ossia al primo o all'ultimo posto del cosiddetto z-order (così chiamato perché si riferisce a un immaginario asse z, perpendicolare agli assi x e y).

```
void dispose()
```

Chiude la Window e le finestre dipendenti, rilasciando tutte le risorse del sistema usate per la visualizzazione della finestra stessa, delle finestre dipendenti e di tutti i componenti contenuti nelle suddette finestre. La finestra può essere nuovamente visualizzata con una chiamata a `show()`.

```
void show()
```

Rende visibile la finestra; se la finestra o l'owner non sono visualizzabili al momento della chiamata, vengono resi visualizzabili; se la finestra è già visualizzata, viene portata in primo piano.

La visualizzazione di una finestra può essere ottenuta anche con il metodo `pack()`, che sarà descritto nella sezione *Layout*.

La classe `Frame`

La classe `Frame`, nelle applicazioni stand alone (cioè non Applet) è quella che viene utilizzata per la finestra principale, a partire dalla quale eventualmente vengono aperte altre finestre dipendenti. I Frame sono anche gli unici oggetti Window che supportano direttamente una menu bar (menu e menu bar in AWT sono componenti speciali al di fuori della gerarchia Component, come si vedrà più avanti). I metodi per la gestione dei menu saranno visti nel paragrafo sui menù. Un Frame ha una barra del titolo, visualizzata secondo lo stile del sistema grafico utilizzato; può essere minimizzata sotto forma di icona, e ha un bordo che può servire anche per consentire all'utente di ridimensionare la finestra per mezzo del mouse.

Di seguito i principali metodi della classe `Frame`.

```
Frame()
```

```
Frame(String title)
```

Il solo paramentro che si puo passare al costruttore è il titolo; un `Frame` non può avere un owner.

```
static Frame[] getFrames()
```

Non avendo owner, una lista dei Frames dell'applicazione può essere ottenuta solo da questo metodo statico.

```
void setTitle(String s)
String getTitle()
```

Per assegnare e restituire il titolo che compare nella barra del titolo.

```
void setIconImage(Image i)
Image getIconImage()
```

Per assegnare e restituire l'icona che rappresenta la finestra in stato minimizzato. L'icona è rappresentata da un oggetto `Image`, di cui si parlerà nella sezione *Grafica*.

```
void setResizable(boolean v)
boolean isResizable()
```

Per determinare se la finestra può essere ridimensionata dall'utente con il mouse.

```
void setState(int state)
int getState()
```

Per impostare e restituire lo stato della finestra. I possibili valori sono `NORMAL` e `ICONIFIED`. Con `setState()` si può quindi minimizzare la finestra o riportarla alle dimensioni normali.

La classe `Dialog`

Un `Dialog` è una finesta top-level dotata di barra del titolo e bordo, utilizzata general-

mente come finestra di interazione con l'utente, soprattutto per l'input di dati e comandi. Un `Dialog` è sempre dipendente da un'altra finestra (tipicamente un `Frame` o un'altro `Dialog`) e quando la owner viene minimizzata o nascosta, lo stesso accade al dialog dipendente.

Quello che caratterizza maggiormente il `Dialog` è la possibilità di funzionare in modalità cosiddetta *modal*. Un modal dialog resta in primo piano e mantiene il focus finché non viene chiusa, o finché non viene attivata un'altra applicazione. Finché il modal dialog è aperto, nessun'altra finestra della stessa applicazione può ottenere il focus o passare in primo piano; quindi di fatto blocca gli altri componenti dell'applicazione finché non viene chiuso. I modal dialog vengono usati per far sì che l'utente non possa continuare determinate operazioni, per loro natura sequenziali, finché non ha concluso altre operazioni preliminari (ad esempio la conferma di un comando, o l'inserimento di una serie di dati).

Questi i principali metodi della classe `Dialog`.

```
Dialog(Frame owner)

Dialog(Dialog owner)
```

L'owner, che deve essere passato al costruttore, può essere un `Frame` o un altro `Dialog`.

```
void setTitle(String s)

String getTitle() come in Frame

void setResizable(boolean v)

boolean isResizable() come in Frame

void setModal(boolean v)

boolean getModal()
```

Assegnano e restituiscono lo stato `modal`.

Un esempio su `Frame` e `Dialog`

Il seguente esempio mostra alcune delle funzionalità di `Frame` e `Dialog`. In particolare si mostrerà come posizionare i componenti con coordinate esplicite, piuttosto che con i layout, come compiere operazioni sui componenti "figli", e la differenza di funzionamento tra modal dialog e non-modal dialog.

```
import java.awt.*;
import java.awt.event.*;

public class NoLayoutFrame extends Frame {
```

```
Label descriptionLabel;
Button componentListButton;
Checkbox modalCheckbox;

public NoLayoutFrame() {
   initFrameProperties();
   createDescriptionLabel();
   createComponentListButton();
   createModalCheckbox();
}

void initFrameProperties() {
   setTitle("Window senza layout");
   setLocation(200, 200);
   setSize(500, 300);
   setLayout(null);
   addWindowListener(new MainWindowListener());
}

void createDescriptionLabel() {
   descriptionLabel = new Label("I componenti in questa finestra "
                              + "sono posizionati con coordinate fisse");
   descriptionLabel.setBounds(30, 50, 450, 20);
   add(descriptionLabel);
}

void createModalCheckbox() {
   modalCheckbox = new Checkbox("Modal Dialog");
   modalCheckbox.setBounds(100, 100, 180, 20);
   add(modalCheckbox);
}

void createComponentListButton() {
   componentListButton = new Button("Lista dei componenti");
   componentListButton.setBounds(100, 160, 180, 40);
   componentListButton.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Component[] components = getComponents();
            int count = components.length;
            StringBuffer list = new StringBuffer();
            for (int i = 0; i < count; i++)
               list.append(components[i].getName() + ": "
                           + components[i].getClass().getName() + c\n');
            new MessageDialog(NoLayoutFrame.this, "Lista componenti",
                              list.toString(), modalCheckbox.getState()).show();
         }
      }
   );
```

```
      add(componentListButton);
   }

   public static void main(String[] args) {
      new NoLayoutFrame().show();
   }
}
```

Il `Frame` contiene una `Label`, che mostra un messaggio statico, un `Button` che apre un `Dialog` in cui compare una lista dei componenti del `Frame`, e un `Checkbox` che determina se il `Dialog` aperto dal `Button` sarà o meno `modal`.

Ciascun componente ha una sua variabile di classe che ne contiene il reference, in modo che tutti i metodi della classe possano avere accesso ai componenti. L'inizializzazione del frame e quella di ciascun componente sono affidate ognuna a un diverso metodo privato piuttosto che tutte a un unico metodo, che finirebbe con il diventare eccessivamente lungo e confuso. In questo modo il codice relativo a ciascun componente è chiaramente individuato, facilitando la comprensione del codice e la realizzazione di eventuali modifiche implementative. Ogni metodo `createXxx()` crea un componente di un certo tipo e lo assegna alla variabile corrispondente. Poi assegna le proprietà (oltre a quelle passate direttamente al costruttore). In particolare si noti l'uso del metodo `setBounds()` per assegnare posizione e dimensioni. Poiché il frame è senza layout, l'uso di questo metodo è imprescindibile per assicurare che l'oggetto venga visualizzato. Infatti mentre il layout, quando presente, si fa carico di determinare non solo la posizione, ma anche le dimensioni del componente, in sua assenza le dimensioni iniziali sono nulle, il che significa che l'oggetto non è visibile. Infine, ogni componente viene inserito nel `Frame` con il metodo `add()`.

Nel metodo `initFrameProperties()` viene aggiunto un `WindowListener` che gestisce l'evento di chiusura della finestra, come nell'esempio precedente. Questa volta però il listener è definito come classe pubblica separata dalla classe. Infatti implementa un comportamento molto comune nelle finestre principali, rendendo funzionante il pulsante di sistema della title bar per la chiusura della finestra e determinando, alla chiusura della finestra, l'uscita dall'applicazione. Poiché il comportamento in questione è condiviso da molte applicazioni, risulta naturale e proficuo implementare la classe in modo indipendente per consentirne il riuso.

```
import java.awt.event.*;

public class MainWindowListener extends WindowAdapter {
   public void windowClosing(WindowEvent e) {
      e.getWindow().dispose();
   }
```

```
   public void windowClosed(WindowEvent e) {
      System.exit(0);
   }
}
```

L’evento `WindowClosing`, generato ad ogni *richiesta* di chiusura della finestra, è quello causato (tra l’altro) da un click sul pulsante di chiusura sulla title bar. Il metodo `windowClosing()` non fa altro che chiamare il metodo `dispose()` della finestra che ha generato l’evento, per chiuderla effettivamente.

L’evento `WindowClosed`, generato invece ad ogni effettiva chiusura della finestra, è gestito come nel precedente esempio, determinando la terminazione dell’applicazione.

Tornando al `NoLayoutFrame`, si può osservare che il layout di default, creato automaticamente dal costruttore del `Frame`, viene “buttato via” tramite l’istruzione `setLayout(null)` nel metodo `initFrameProperties()`. Nello stesso metodo si può notare l’uso dei metodi `setLocation()` e `setSize()` in alternativa a `setBounds()`.

Nel metodo `createComponentListButton()` si trova un esempio d’uso di una *classe anonima*. In Java si può definire una classe “al volo”, anche all’interno di un metodo, utilizzando la sintassi mostrata nell’esempio; la classe non ha un nome, ma viene definita come `new typeName`, dove typeName è il nome della classe che viene estesa o dell’interfaccia che viene implementata.

Nell’esempio la classe anonima è utilizzata per definire il `Listener`, e quindi il modo di rispondere agli eventi, nello stesso metodo che crea e inizializza il componente. Una simile soluzione può essere opportuna, o comunque accettabile, solo in casi come quello mostrato, in cui il codice è molto semplice e breve. In casi più complessi, anche se si usa una classe anonima, è consigliabile separare le due implementazioni, in questo modo:

```
ActionListener componentListButtonListener = new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      Component[] components = getComponents();
      int count = components.length;
      StringBuffer list = new StringBuffer();
      for (int i = 0; i < count; i++)
         list.append(components[i].getName() + ": "
                     + components[i].getClass().getName() + '\n');
      new MessageDialog(NoLayoutFrame.this, "Lista componenti",
                        list.toString(), modalCheckbox.getState()).show();
   }
}

void createComponentListButton() {
   componentListButton = new Button("Lista dei componenti");
   componentListButton.setBounds(100, 160, 180, 40);
   componentListButton.addActionListener(componentListButtonListener);
```

```
   add(componentListButton);
}
```

L'implementazione dei listener è un argomento abbastanza complesso, su cui si tornerà nella sezione *Gestione degli eventi*.

Nel metodo `actionPerformed()` del listener viene costruito uno `StringBuffer` che contiene una lista dei componenti (per nome e tipo), utilizzando il metodo `getComponents()` del `Frame`. La lista viene passata al costruttore di un oggetto `MessageDialog` che viene creato e visualizzato (metodo `show()`) e che a sua volta visualizza la lista. Allo stesso costruttore viene passato anche un argomento boolean, dipendente dallo stato del `Checkbox`, che determina il tipo di `Dialog` (modal o non-modal).

Questa è la definizione della classe `MessageDialog`:

```
import java.awt.*;
import java.awt.event.*;

public class MessageDialog extends Dialog {
   String message;
   TextArea messageText;
   Button closeButton;

   public MessageDialog(Frame owner, String title, String message,
                        boolean modal) {
      super(frame, title, modal);
      setBounds(100, 100, 300, 300);
      setLayout(new BorderLayout());
      createMessageText(message);
      createCloseButton();
   }

   public void createMessageText(String message) {
      messageText = new TextArea(message, 0, 0, SCROLLBARS_NONE);
      messageText.setEditable(false);
      add(messageText, BorderLayout.CENTER);
   }

   public void createCloseButton() {
      closeButton = new Button("Chiudi");
      closeButton.addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               dispose();
            }
         }
      );
      add(closeButton, BorderLayout.SOUTH);
```

```
    }
}
```

Questo dialogo contiene soltanto una `TextArea` in cui viene visualizzato il messaggio passato in argomento al costruttore e un pulsante che chiude il `Dialog` con il metodo `dispose()`. Il costruttore prende anche il `Frame owner`, il titolo, e un flag che indica se il dialogo deve essere di tipo modal. Questi argomenti vengono passati al costruttore della classe base `Dialog`, mentre il messaggio viene passato al metodo che crea la `TextArea`.

Se viene eseguito il `main()`, si può notare il diverso comportamento del modal dialog rispetto a quello non-modal: quest'ultimo, una volta aperto, non impedisce all'utente di continuare a interagire con il Frame; se il Frame viene chiuso o ridotto a icona, anche il dialog viene chiuso (temporaneamente nel secondo caso).

Il modal dialog invece monopolizza l'input di tastiera e mouse, e l'utente non può agire sul Frame finché il dialogo non viene chiuso.

Visualizzando l'elenco dei componenti si può notare inoltre che il *nome* dei componenti, se non esplicitamente assegnato, ha come valore una stringa di default generata dalla Virtual Machine.

Contenitori non-window

Le classi `Panel` e `ScrollPane`, rappresentano dei contenitori che non sono finestre, ma possono soltanto essere inserite all'interno di un altro contenitore.

La classe `Panel` non aggiunge nulla all'interfaccia di un generico `Container`, da cui deriva. La differenza è che con un Panel viene creato un peer nell'ambiente grafico sottostante.

La classe `ScrollPane` deriva anch'essa direttamente da `Container`, quindi non è un tipo di `Panel`. Contiene due componenti `ScrollBar` (orizzontale e verticale) che consentono al pannello di contenere un'area di dimensioni superiori a quella effettivamente visualizzata (chiamata *viewport*); le scroll bar permettono di visualizzare le porzioni nascoste, "spostando" il viewport. Il comportamento di uno `ScrollPane` è caratterizzato da una *display policy*, un parametro che può assumere uno dei seguenti valori:

`SCROLLBAR_AS_NEEDED`

Questo è il valore di default; le `ScrollBar` vengono create e visualizzate solo se necessario, ossia se la corrispondente dimensione, orizzontale o verticale, dell'area contenuta eccede quella del contenitore.

`SCROLLBAR_ALWAYS`

Le `ScrollBar` vengono create e visualizzate comunque.

`SCROLLBAR_NEVER`

Le `ScrollBar` non vengono create. In questo caso lo scrolling dell'area può essere effettuato da codice con il metodo `setScroolPosition()`, implementando un sistema di scrolling personalizzato.

Lo `ScrollPanel` mantiene un suo controllo del layout, per cui il metodo `setLayout()` non può essere utilizzato. Di conseguenza, nella maggioranza dei casi, si dovrà inserire un altro container (Panel o sottoclasse) all'interno dello `ScrollPane` per poter determinare il layout.

Interfacce dinamiche

Il seguente esempio mostra l'uso di uno `ScrollPane`, nel suo funzionamento di default, in cui le scrollbar vengono create e modificate a runtime, secondo le necessità. Questa modalità di funzionamento è un caso di interfaccia *di tipo dinamico*, ossia con elementi che vengono generati e/o modificati nel corso dell'esecuzione. Normalmente le interfacce grafiche vengono costruite in maniera statica, ossia ogni finestra ha i suoi componenti costruiti e inseriti in fase di inizializzazione, che rimangono costanti durante il funzionamento sia nell'aspetto che nel comportamento. L'aspetto dinamico è dato dall'interazione con l'utente e da quella tra i vari componenti. Quest'impostazione è condizionata spesso dall'uso di tools grafici che permettono di costruire le interfacce in maniera “visuale”, con il sistema del drag and drop, ma non offrono facilitazioni per la costruzione di elementi dinamici; quindi si tende spesso a dimenticare l'esistenza di queste possibilità. Il seguente codice, oltre a mostrare il funzionamento dinamico dello `ScrollPane`, è anche un esempio, per quanto rudimentale, di un'interfaccia essa stessa impostata in maniera dinamica.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class FileReaderFrame extends Frame {
   Panel textPanel;
   ScrollPane scrollPane;
   Button continueButton = new Button("Continua");

   class ReaderListener implements ActionListener {
      BufferedReader reader;

      ReaderListener(String fileName) {
         try {
            reader = new BufferedReader(new FileReader(fileName));
         } catch (FileNotFoundException e) {
            showError(e);
```

```
            }
        }

        public void actionPerformed(ActionEvent event) {
            try {
                String line = reader.readLine();
                if (line != null)
                    addTextLine(line);
            } catch (IOException e) {
                showError(e);
            }
        }
    }

    public FileReaderFrame(String title, String message, String fileName) {
        super(title);
        setBounds(100, 100, 260, 200);
        addWindowListener(new MainWindowListener());
        createTextPanel();
        createCommandPanel();
        start(message, fileName);
    }

    void createTextPanel() {
        scrollPane = new ScrollPane();
        textPanel = new Panel();
        scrollPane.add(textPanel);
        textPanel.setLayout(new GridLayout(0, 1));
        add(scrollPane, BorderLayout.CENTER);
    }

    Panel createCommandPanel() {
        Panel p = new Panel();
        p.add(continueButton);
        p.add(new CloseButton(this));
        add(p, BorderLayout.SOUTH);
    }

    void start(String message, final String fileName) {
        textPanel.add(new Label(message));
        continueButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    textPanel.removeAll();
                    continueButton.removeActionListener(this);
                            continueButton.addActionListener(new
                                ReaderListener(fileName));
                }
            }
```

```
        );
    }

    void addTextLine(String line) {
        textPanel.add(new Label(line));
        textPanel.setSize(textPanel.getWidth(),
                          textPanel.getComponentCount() * 20);
        double viewportHeight
        = scrollPane.getViewportSize().getHeight() - 60;
        double heightDiff = textPanel.getHeight() - viewportHeight;
        scrollPane.validate();
        if (heightDiff > 0)
            scrollPane.setScrollPosition(0, (int)(heightDiff));
    }

    void showError(Exception e) {
        new MessageDialog(this, "Errore", e.getMessage(), true);
    }

    public static void main(String[] args) {
        Frame frame
        = new FileReaderFrame("La vispa Teresa",
                              "Premi 'Continua' se vuoi leggere la poesia",
                              "LaVispaTeresa.txt");
        frame.show();
    }
}
```

Il costruttore del `Frame` prende tre argomenti: il titolo, un messaggio e il nome di un file. All'apertura, il frame viene visualizzato con il titolo assegnato e mostra il messaggio iniziale. Premendo il pulsante `Continua` il messaggio viene cancellato e comincia la lettura del file, una riga alla volta, ad ogni azione del pulsante `Continua`.

Il `Frame` contiene uno `ScrollPane` che a sua volta contiene un `Panel (textPanel)`, utilizzato per la visualizzazione del testo, per mezzo di una serie di `Label`. Vi è inoltre un altro Panel (`commandPanel`), che contiene i due pulsanti di comando `Continua` e `Chiudi`.

Nel metodo `start()`, chiamato dopo la creazione e l'inizializzazione dei componenti, viene inserita in `textPanel` una `Label` che visualizza un messaggio passato originariamente al costruttore. Dopodiché viene definito e assegnato un listener al pulsante `Continua` (`continueButton`); questo introduce un primo comportamento "dinamico": questo listener è concepito per funzionare una volta sola; esso rimuove il messaggio dal `textPanel`, dopodiché rimuove se stesso dalla lista dei listener di `continueButton` e inserisce invece un altro listener, di tipo `ReaderListener`, definito come inner class del Frame. Da questo momento in poi un click su `continueButton` provocherà la lettura e la visualizzazione, tramite una `Label`, di una riga di testo del file specificato.

Le `Label` vengono create a runtime di volta in volta, finché non si raggiunge la fine del file, e vengono inserite all'interno del `Panel`, che a sua volta sta all'interno dello `ScrollPane`. Ogni volta che viene aggiunta una `Label`, il pannello viene ridimensionato. Quando l'altezza del pannello supera quella dello `ScrollPane`, compare automaticamente la scroll bar verticale.

Il metodo `addTextLine()` è quello che aggiunge una `Label` ogni volta (chiamato dal `ReaderListener` di `continueButton`). L'altezza del `textPanel` viene ogni volta impostata in modo che ogni label abbia uno spazio di 20 pixel. Il metodo poi provvede a modificare la posizione dell'area visualizzata, tramite `setScrollPosition()`, in modo che sia sempre visualizzata l'ultima riga letta: la "scroll position" indica le coordinate, all'interno all'area totale visualizzabile (corrispondente in questo caso al `textPanel`), del punto che viene posizionato all'origine della viewport. Per far sì che venga visualizzata tutta la parte inferiore del pannello, la scroll position dovrebbe essere uguale alla differenza tra l'altezza del pannello e quella del viewport. In pratica è stato necessario inserire una costante correttiva di 60 pixel, apparentemente per qualche bug interno dello `ScrollPanel`.

Al `textPanel` viene asegnato un tipo di layout diverso da quelli finora utilizzati, il `GridLayout`. Questo layout divide l'area del container in una serie di celle di uguale dimensione; al momento della creazione vengono specificati il numero di righe e colonne, che determinano il numero e la collocazione delle celle. Se il numero di righe o colonne specificato è uguale `0` (non possono esserlo tutti e due) il layout è dinamico: il valore iniziale è `1`, ma può cambiare a runtime se vengono aggiunti altri componenti in numero superiore a quello delle celle create inizialmente: in tal caso verrà aggiunta una nuova riga o colonna (o verrà tolta in caso disuccessiva rimozione dei componenti). Nel caso specifico si è creato `GridLayout` è di una colonna con numero di righe variabile dinamicamente.

Poiché la visualizzazione delle `Label` è controllata dal `GridLayout`, inizialmente le label compaiono distanziate tra loro, distribuendosi in tutta l'area del pannello, la cui dimensione iniziale è uguale a quella dello `ScrollPane`. Solo quando la dimensione del pannello raggiunge e supera quella dello `ScrollPane`, le `Label` compaiono ravvicinate e la distanza tra le righe si mantiene costante. Purtroppo i layout di AWT sono generalmente poco controllabili e tendono ad applicare un criterio predefinito e poco flessibile nella visualizzazione dei componenti. Visualizzare correttamente le label partendo dalla parte superiore del Panel e mantenendo la distanza costante avrebbe complicato eccessivamente il codice, per cui si è preferito ignorare per ora il problema. Si tornerà sull'argomento nella sezione *Layout*.

Sempre in `addTextLine()`, si noti l'uso di `validate()`, necessario per aggiornare il layout dei componenti, ogni volta che se ne aggiunge uno.

Si notino infine alcune scelte di design che incrementano la riusabilità del codice realizzato: l'aver definito precedentemente una classe `MainWindowListener` che gestisce gli eventi legati alla chiusura della finestra ha permesso di riutilizzarla anche in questo esem-

pio. Qui è stato individuato un altro componente potenzialmente riusabile: il pulsante di chiusura `CloseButton`, che per questo motivo è stato definito come classe a sé stante.

```
import java.awt.*;
import java.awt.event.*;

public class CloseButton extends Button {
   Window window;

   public CloseButton(Window win) {
      this(win, null);
   }

   public CloseButton(Window win, String text) {
      window = win;
      if (text == null)
         text = "Chiudi";
      setLabel(text);
      addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               window.dispose();
            }
         }
      );
   }
}
```

Il componente ha un comportamento identico a quello usato nella classe `MessageDialog` del precedente esempio. Per poter essere usato con qualunque Window, viene definita una variabile di classe che mantiene un reference alla Window, passato come argomento del costruttore. Anche l'etichetta del pulsante è parametrizzata, per ottenere una maggiore flessibilità.

Come esempio di interfaccia dinamica, quello appena presentato non è particolarmente significativo ed è giustificato solo dalle sue finalità didattiche: a parte la discutibile utilità di visualizzare una riga alla volta, si sarebbe potuto ottenere lo stesso risultato molto più semplicemente con un singolo componente `TextArea`. Ma si pensi a casi più complessi, come un questionario, ad esempio un'indagine di mercato, in cui la domanda successiva viene scelta a seconda della risposta a quella precedente (a seconda dell'età, della professione, degli hobby…). In un caso del genere sarebbe possibile, con un sistema simile a quello mostrato, costruire passo per passo un questionario personalizzato, creando a runtime i componenti opportuni, che possono anche essere di diverso tipo (`TextField`, `Checkbox`, `Choice`, ecc.) a seconda delle domande via via proposte.

Visualizzazione e editing di testi

Questa e le prossime sezioni sono dedicate a una descrizione dei componenti AWT che non sono contenitori di altri componenti.

Il primo gruppo è quello dei componenti con cui si può visualizzare, inserire, modificare un testo.

`Label`

Questo è il componente più semplice, ed è tendenzialmente un componente statico. Non è selezionabile e non è modificabile dall'utente. Le sue sole proprietà particolari sono il testo (disposto sempre su una sola riga) e l'allineamento, accessibili attraverso i metodi

```
void setText(String s)

String getText()
```

per il testo, e

```
void setAlignment(int alignment)

int getAlignment()
```

per l'allineamento orizzontale.

I valori possibili sono le costanti `LEFT`, `RIGTH`, `CENTER`, e determinano come viene disposto il testo quando le dimensioni del componente sono superiori a quelle del testo.

`TextComponent`

`TextComponent` è una classe che, pur non essendo formalmente astratta, serve solo da classe base per i due componenti `TextField` e `TextArea`. La sua interfaccia comprende i metodi comuni a entrambi i componenti, che sono qui elencati divisi per categoria.

Accesso al testo:

```
void setText(String s)

String getText()
```

Modificabilità:

```
void setEditable(boolean b)

boolean getEditable()
```

Se il valore è `false`, il componente non riceve input da tastiera.

Quanto al cursore di testo, i componenti derivati da `TextComponent` visualizzano un cursore di testo (*caret*) nella posizione di inserimento; i due metodi:

```
void setCaretPosition(int pos)
int getCaretPosition
```

permettono di modificare o rilevare il valore di questa posizione, espresso come l'indice (a partire da `0`) del carattere che si trova subito dopo il caret.

Per la selezione del testo, il testo all'interno del componente è selezionabile dall'utente per intero o in parte, con modalità dipendenti dal sistema grafico, ma comunque sempre per mezzo della tastiera e del mouse. La selezione può anche essere effettuata da programma, con alcuni dei metodi qui descritti.

```
String getSelectedText()
```

Restituisce la porzione di testo selezionata.

```
void setSelectionStart(int pos)
void setSelectionEnd(int pos)
select(int start, int end)
```

Determinano la posizione d'inizio e fine del testo selezionato; la prima posizione è 0. Se vengono specificati valori errati (valori negativi, fine prima dell'inizio, valori oltre la lunghezza totale del testo, ecc.) non viene generato nessun errore, ma i valori vengono corretti automaticamente.

```
int getSelectionStart()
int getSelectionEnd()
```

Restituiscono le posizioni di inizio e fine del testo selezionato.

`TextField`

Il componente `TextField` rappresenta una casella di testo editabile di una sola riga. La lunghezza del testo immesso può essere superiore alla dimensione orizzontale del componente. In tal caso il testo può essere fatto scorrere agendo sulla tastiera. Non esiste una proprietà che consente di limitare il numero di caratteri inseribili. Per ottenere un simile

controllo è ecessario gestire gli eventi generati dall'inserimento del testo (si veda la sezione *Eventi*).

Oltre ai metodi di `TextComponent`, `TextField` ha alcuni metodi suoi propri.

Per i costruttori, oltre a quello di default si possono usare:

```
TextField(String text)
```

che assegna un testo predefinito,

```
TextField(int columns)
```

che assegna una dimensione in colonne (vedere sotto),

```
TextField(String text, int columns)
```

che assegna entrambe le proprietà.

Dimensione in colonne:

```
void setColumns(int number)
int getColumns()
```

Permettono di assegnare e rilevare la dimensione orizzontale (width) non in pixel ma in "colonne", che rappresentano la larghezza media di un carattere del font attualmente in uso da parte del componente.

Echo character ovvero "carattere d'eco" è un carattere che, se assegnato, viene visualizzato al posto di ciascun carattere inserito nel componente; viene generalmente utilizzato per nascondere il testo della password in un campo di input, visualizzando però un' "eco" dei caratteri inseriti. Normalmente si usa l'asterisco.

```
void setEchoChar(char c)
char getEchoChar()
```

Sono i metodi di accesso.

```
boolean echoCharIsSet()
```

Indica se il carattere è assegnato (e quindi se il testo è “mascherato”).

`TextArea`

Una TextArea è una casella di testo con un numero arbitrario di righe, e con delle scrollbar opzionali, gestite automaticamente. Questi sono i metodi peculiari della TextArea.

Costruttori

`TextArea()`

Costruttore di default.

`TextArea(String text)`

Assegna un testo predefinito.

`TextArea(int rows, int columns)`

Assegna una dimensione in righe e colonne.

`TextArea(String text, int rows, int columns)`

Una combinazione dei precedenti

`TextArea(String text, int rows, int columns, int scrollbars)`

Aggiunge una costante che influenza la visualizzazione delle scrollbar (vedere sotto).

Dimensione in righe e colonne

`void setRows(int number)`

`int getRows()`

Assegnano e restituiscono la dimensione verticale in righe, corrispondenti all’altezza dei caratteri nel font corrente.

```
void setColumns(int number)
int getColumns()
```

Assegnano e restituiscono la dimensione orizzontale in colonne.

Per l'inserimento testo, poiché una `TextArea` ha normalmente un testo più lungo di quello di un `TextField`, sono presenti altri metodi per rendere più flessibile l'inserimento del testo, anche da programma (nel `textField` si può solo sostituire l'intero testo con `setText()`).

```
void append(String s)
```

Aggiunge la stringa specificata alla fine del testo.

```
void insert(String s, int pos)
```

Inserisce la stringa nella posizione specificata.

```
void replaceRange(String s, int start, int end)
```

Sostituisce il testo compreso tra le posizioni specificate, con la stringa data.

Scrollbar

```
int getScrollbarVisibility()
```

Restituisce una costante che indica lo stato di visibilità delle scrollbar; può assumere uno dei seguenti valori: `SCROLLBARS_BOTH` quando le scrollbar sono entrambe presenti, `SCROLLBARS_HORIZONTAL_ONLY` quando solo la scrollbar orizzontale è presente, `SCROLLBARS_VERTICAL_ONLY` quando solo la scrollbar verticale è presente, `SCROLLBARS_NONE` quando non compare nessuna delle due scrollbar.

La visualizzazione delle scrollbar è fissa, ossia indipendente dalle dimensioni del testo e determinata una volta per tutte nel costruttore. Il valore di default è `SCROLLBARS_BOTH`. Non esiste un metodo `setScrollbarVisibility()` che permetta di modificare la visualizzazione delle scrollbar dopo la creazione del componente.

Pulsanti

Button

La classe `Button` rappresenta un pulsante di comando, la cui unica funzione è quella di eseguire un'azione quando viene "azionato" tramite un click o con la tastiera.

Sono rappresentati da rettangoli con un testo all'interno, disposto al centro dell'area del componente, su un'unica riga. Questi i principali metodi:

```
void setLabel(String s)

String getLabel()
```

che sono metodi di accesso al testo visualizzato, e

```
void setActionCommand(String s)

String getActionCommand()
```

i quali danno accesso al nome del comando, un parametro che per default è uguale al testo della label, che può essere utilizzato nella gestione di un evento `ActionEvent` (si veda la sezione *Eventi*).

`Checkbox`

La classe `Checkbox` rappresenta elementi grafici selezionabili; anzi, per essere più precisi, *caratterizzati* dal loro stato "selezionato" o "non selezionato". In altre parole la funzione per cui sono concepiti è quella di indicare lo stato di una certa proprietà (attiva o non attiva). L'altra caratteristica essenziale di questi componenti è la possibilità di essere inseriti in *gruppi* che rappresentano una serie di *scelte mutuamente esclusive*. In questa forma sono chiamati in alcuni sistemi grafici *radio button*, così chiamati perché funzionano come i pulsanti di preselezione del canale di una radio: nel momento in cui uno viene premuto, quello precedentemente selezionato torna automaticamente alla posizione "off". Anche se in certe implementazioni i peer sono componenti differenti a seconda che si tratti di normale checkbox o di radio button, la classe li rappresenta entrambi. Quello che modifica il loro comportamento, e in genere anche il loro aspetto, è l'essere o meno inseriti in un oggetto `CheckboxGroup` (anche questa classe sarà descritta tra breve).

Ecco una lista dei principali metodi della classe.

Costruttori

```
CheckBox()
```

Costruttore di default.

```
CheckBox(String label)
```

Assegna l'etichetta del checkbox.

```
CheckBox(String label, boolean state)
```

Assegna etichetta e stato.

```
CheckBox(String label, boolean state, CheckBoxGroup group)
```

Assegna etichetta, stato e gruppo

```
CheckBox(String label, CheckBoxGroup group, boolean state)
```

Come il precedente, ma con diverso ordine di argomenti. Tutti gli argomenti accettano il valore `null`, che equivale a una mancata assegnazione della corrispondente proprietà.

I metodi di accesso all'etichetta, un testo che viene visualizzato in genere a fianco (sulla destra) del pulsante, sono

```
void setLabel(String s)
String getLabel()
```

Stato

```
void setState(boolean checked)
boolean getState()
```

Accedono allo stato (`true` per selezionato, `false` per non selezionato).

```
void setCheckboxGroup(CheckboxGroup group)
CheckboxGroup getCheckboxGroup()
```

Se si assegna un gruppo ad un `Checkbox` che già appartiene a un altro gruppo, il `Checkbox` viene rimosso dal precedente gruppo. Inserire il checkbox in un gruppo può determinare un cambiamento nel modo in cui l'oggetto viene visualizzato. La classe `CheckboxGroup` ha il solo costruttore di default e contiene i metodi

```
void setSelectedCheckbox(Checkbox c)
Checkbox getSelectedCheckbox()
```

Quest'ultimo restituisce `null` se nessun `Checkbox` è selezionato. `CheckboxGroup` non è un componente e non ha una rappresentazione grafica, è solo una classe di supporto che serve per raggruppare i checkbox in un insieme di radio buttons.

Un esempio su checkbox e componenti di testo

Il seguente esempio mostra il funzionamento dei `Checkbox`, nelle due modalità normale e "radio button", e dei `TextComponent` (`TextField` e `TextArea`). I `Checkbox` sono utilizzati per modificare le proprietà del `TextComponent` visualizzato.

```
import java.awt.*;
import java.awt.event.*;

public class TextComponentDemo extends Frame {
   final String TEXT_AREA = TextArea.class.getName();
   final String TEXT_FIELD = TextField.class.getName();
   Panel textPanel;
   Panel configPanel;
   Panel commandPanel;
   TextComponent textComponent;
   Checkbox horizontalScrollCBox;
   Checkbox verticalScrollCBox;
   Checkbox textAreaCBox;
   Checkbox textFieldCBox;
   CheckboxGroup componentTypeCBGroup;

   public TextComponentDemo() {
      createTextPanel();
      createConfigPanel();
      createCommandPanel();
      createComponentTypeCheckboxes();
      createScrollCheckboxes();
      createTextComponent(TEXT_FIELD);
      commandPanel.add(new CloseButton(this));
      addWindowListener(new MainWindowListener());
      setSize(300, 200);
      setTitle("TextComponent Demo");
   }

   // l'implementazione dei metodi verrà mostrata in seguito
   ...

   public static void main(String[] args) {
      TextComponentDemo frame = new TextComponentDemo();
      frame.show();
   }
}
```

Da questo primo frammento di codice si può già vedere la struttura generale della classe. Il `Frame` contiene tre `Panel`: il primo conterrà il `TextComponent`, il secondo i `Checkbox` che modificheranno la "configurazione" del `TextComponent`, il terzo il pul-

sante di chiusura. La variabile `TextComponent` è di un tipo generico e farà riferimento a un oggetto di tipo `TextField` oppure `TextArea`, a seconda di quale dei due `Checkbox` `textAreaCBox` e `textFieldCBox` viene selezionato.

Il `TextComponent` viene creato dal metodo `createTextComponent`, che prende come argomento il nome della classe da istanziare, ossia il valore di una delle due costanti `TEXT_AREA` o `TEXT_FIELD`. Inizialmente, come si vede, la classe è `TextField`.

Gli altri due `Checkbox` controllano invece la visualizzazione delle `Scrollbar`, orizzontale e verticale, nel caso il componente di testo sia una `TextArea`.

Il `commandPanel` contiene un pusante `CloseButton`, istanza della classe precedentemente definita. Il `Frame` utilizza anche questa volta un listener di tipo `MainWindowListener`.

Quelli che seguono sono i metodi che creano i tre contenitori:

```
void createTextPanel() {
   textPanel = new Panel();
   textPanel.setLayout(new BorderLayout());
   add(textPanel, BorderLayout.CENTER);
}


void createConfigPanel() {
   configPanel = new Panel();
   configPanel.setLayout(new GridLayout(0, 1));
   add(configPanel, BorderLayout.EAST);
}


void createCommandPanel() {
   commandPanel = new Panel();
   add(commandPanel, BorderLayout.SOUTH);
}
```

Al `textPanel` viene assegnato un `BorderLayout` per i motivi che si vedranno in seguito, mentre il `configPanel` ha un `GridLayout` di una sola colonna e numero indefinito di righe. Il `commandPanel` mantiene invece il `FlowLayout` di default.

Il prossimo metodo è quello che crea i Checkbox che selezionano il tipo di componente di testo:

```
void createComponentTypeCheckboxes() {
   ItemListener listener = new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
         createTextComponent(((Component)e.getSource()).getName());
      }
```

```
   };
   componentTypeCBGroup = new CheckboxGroup();
   textFieldCBox = new Checkbox("TextField", componentTypeCBGroup, true);
   textFieldCBox.setName(TEXT_FIELD);
   textFieldCBox.addItemListener(listener);
   textAreaCBox = new Checkbox("TextArea", componentTypeCBGroup, false);
   textAreaCBox.setName(TEXT_AREA);
   textAreaCBox.addItemListener(listener);
   configPanel.add(textFieldCBox);
   configPanel.add(textAreaCBox);
}
```

Prima dei checkbox viene creato il `CheckboxGroup componentTypeCBGroup`, che poi viene passato al costruttore dei due `Checkbox`. In tal modo i `Checkbox` apparterranno allo stesso gruppo e si comporteranno da radio button, cioè solo un componente del gruppo può essere selezionato in un dato momento. Questo corrisponde al funzionamento logico dei Checkbox: infatti il tipo di `TextComponent` potrà essere solo o `TextField` o `TextArea`, e non tutti e due assieme.

Il listener, di tipo `ItemListener`, viene definito e creato all'interno del metodo e assegnato a entrambi i Checkbox. La risposta all'evento di selezione e una chiamata al metodo `createTextComponent()`, passando come argomento il *nome* del componente che ha dato origine all'evento. Qui si può vedere un possibile uso del nome di un componente: facendo corrispondere il nome dei Checkbox a quello della classe dell'oggetto da creare, si fornisce al listener direttamente il parametro da passare al metodo di creazione. Come si vedrà tra poco, il metodo `CreateTextComponent()` si incarica di controllare lo stato dei `Checkbox` per ricreare un componente conforme alle scelte correnti.

Si noti che non viene fatto alcun controllo sul tipo di evento (selezione o deselezione) perché i Checkbox appartengono a un gruppo, e in tal caso ad ogni selezione (corrispondente in effetti a una deselezione automatica del precedente Checkbox selezionato, più una selezione di un nuovo Checkbox) dà origine a un unico evento che è sempre di selezione.

Ecco ora il metodo che crea i `Checkbox` che controllano la visualizzazione delle `Scrollbar` nella `TextArea`:

```
void createScrollCheckboxes() {
   ItemListener listener = new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
         createTextComponent(TEXT_AREA);
      }
   };
   verticalScrollCBox = new Checkbox("Scrollbar verticale");
   verticalScrollCBox.addItemListener(listener);
   horizontalScrollCBox = new Checkbox("Scrollbar orizzontale");
```

```
    horizontalScrollCBox.addItemListener(listener);
    configPanel.add(verticalScrollCBox);
    configPanel.add(horizontalScrollCBox);
    showScrollCheckboxes(false);
}

void showScrollCheckboxes(boolean visible) {
    verticalScrollCBox.setVisible(visible);
    horizontalScrollCBox.setVisible(visible);
}
```

In questo caso i due Checkbox funzionano in modalità normale, ossia non come radio button; ciascuno rappresenta lo stato di visualizzazione di una delle due Scrollbar. Se il Checkbox è selezionato, la Scrollbar corrispondente verrà visualizzata, altrimenti verrà nascosta. Il Listener, di tipo `ItemListener`, viene definito e creato all'interno del metodo e assegnato a entrambi i Checkbox. L'azione in risposta dell'evento di selezione o deselezione è anche in questo caso una semplice chiamata al metodo `createTextComponent()`, mediante il quale viene ricreata la `TextArea` (questi `Checkbox` funzionano soltanto con la `TextArea`, dato che il `TextField` non è provvisto di `Scrollbar`). Questo risulta necessario per il fatto che le impostazioni relative alle Scrollbar possono essere date alla `TextArea` solo nel costruttore, e non possono più essere modificate per un dato oggetto. I Checkbox vengono visualizzati soltanto quando il `TextComponent` è una `TextArea`. Poiché il `TextComponent` creato inizialmente è un `TextField`, i Checkbox vengono nascosti subito dopo la creazione con la chiamata `showScrollCheckboxes(false)`.

Il metodo seguente è quello che compie tutto il lavoro effettivo, ossia la creazione di un `TextComponent` con caratteristiche corrispondenti alle opzioni indicate dall'argomento e dai Checkbox che controllano le Scrollbar:

```
void createTextComponent(String className) {
    String text = null;
    if (textComponent != null) {
        text = textComponent.getText();
        textPanel.remove(textComponent);
    }
    boolean isTextArea = className.equals(TEXT_AREA);
    if (isTextArea)
        textComponent = createTextArea(text);
    else
        textComponent = new TextField(text);
    textPanel.add(textComponent,
                isTextArea ? BorderLayout.CENTER : BorderLayout.NORTH);
    showScrollCheckboxes(isTextArea);
```

```
    validate();
}

TextArea createTextArea(String text) {
    boolean verticalScroll = verticalScrollCBox.getState();
    boolean horizontalScroll = horizontalScrollCBox.getState();
    int scrollBarVisibility = TextArea.SCROLLBARS_NONE;
    if (verticalScroll && horizontalScroll)
        scrollBarVisibility = TextArea.SCROLLBARS_BOTH;
    else if (verticalScroll)
        scrollBarVisibility = TextArea.SCROLLBARS_VERTICAL_ONLY;
    else if (horizontalScroll)
        scrollBarVisibility = TextArea.SCROLLBARS_HORIZONTAL_ONLY;
    return new TextArea(text, 0, 0, scrollBarVisibility);
}
```

Il metodo `createTextComponent()` crea un `TextField` o una `TextArea` a seconda dell'argomento ricevuto, copiando il testo del componente preesistente (se esiste). Se il componente è un `TextField`, questo viene posizionato nella parte superiore del `textPanel`, se invece si tratta di una `TextArea`, viene messa al centro e occuperà l'intera area del Pannello (questo è il funzionamento del `BorderLayout`, come si vedrà meglio in seguito).

La `TextArea` non viene creata direttamente ma per mezzo del metodo `createTextArea()`, che controlla lo stato dei Checkbox per determinare quali Scrollbar devono essere visualizzate, e passa al costruttore il parametro appropriato.

Liste selezionabili

Finora si è descritto un solo tipo di componente selezionabile, il Checkbox. Nel Checkbox ogni opzione è rappresentata da un diverso componente. Invece nel caso dei componenti `List` e `Choice`, il componente rappresenta l'intero insieme di opzioni che sono contenute come elementi (*item*) al suo interno.

`List`

Il componente `List` visualizza una lista di stringhe, ciascuna delle quali costituisce un elemento, in un'area dotata di Scrollbar verticale. La Scrollbar non è controllabile dai metodi della classe, quindi la visualizzazione è gestita dal codice nativo e dal sistema grafico. In genere la Scrollbar viene visualizzata soltanto se necessario.

La lista ha due modalità di funzionamento: a *selezione singola* oppure a *selezione multipla*, simili rispettivamente al funzionamento dei Checkbox inseriti in un `CheckboxGroup`

e a quello dei Checkbox non appartenenti ad un gruppo. Con la selezione singola, ogni volta che un elemento viene selezionato (generalmente con un clic) la selezione viene tolta dall'elemento precedentemente selezionato; con la selezione multipla ogni elemento può essere selezionato e deselezionato indipendentemente dagli altri e ogni clic fa passare l'elemento da uno stato all'altro.

Questi sono i principali metodi della classe `List`.

Costruttori

```
List()
```

Crea una lista con lo spazio per 4 elementi, ossia con quattro righe visibili (la dimensione effettiva dipende dal font corrente) in modalità a selezione singola. La dimensione orizzontale non è determinata.

```
List(int rows)
```

Crea una lista con rows (righe) visibili in modalità a selezione singola.

```
List(int rows, boolean multipleMode)
```

Crea una lista con righe visibili nella modalità indicata dal secondo argomento (se `true` selezione multipla, altrimenti singola)

Manipolazione degli elementi

```
void add(String item)
```

Aggiunge un elemento in coda all'array.

```
void add(String item, int index)
```

Inserisce un elemento nella posizione specificata.

```
void remove(int index)
```

Rimuove l'elemento di dato indice.

`void remove(String item)`

Rimuove l'elemento uguale alla stringa data.

`void removeAll()`

Rimuove tutti gli elementi.

`int getItemCount()`

Restituisce il numero di elementi.

`String[] getItems()`

Restituisce un array contenente tutti gli elementi

`String getItem(int index)`

Restituisce l'elemento di dato indice.

`void replaceItem(String item, int index)`

Sostituisce l'elemento di dato indice con la stringa data.

Selezione

`boolean isMultipleMode()`

Indica se la modalità multipla è attiva.

`void setMultipleMode(boolean b)`

Imposta la modalità di funzionamento a selezione singola o multipla.

`void select(int index)`

`void deselect(int index)`

Selezionano o deselezionano un elemento di dato indice.

`boolean isIndexSelected(int index)`

Indica se l'elemento di dato indice è selezionato.

`int getSelectedIndex()`

Restituisce l'indice dell'elemento selezionato; se nessun elemento è selezionato, oppure se più elementi sono selezionati, restituisce `-1`.

`int[] getSelectedIndexes()`

Restituisce un array contenente gli indici degli elementi selezionati.

`String getSelectedItem()`

Restituisce l'elemento selezionato; se nessun elemento è selezionato, oppure se più elementi sono selezionati, restituisce `null`.

`String[] getSelectedItems()`

Restituisce un array contenente gli elementi selezionati.

Altri metodi

`int getRows()`

Restituisce il numero di righe visibili..

`void makeVisible(int index)`

Fa uno scrolling della lista in modo che l'elemento di dato indice risulti visibile.

```
int getVisibleIndex()
```

Restituisce l'indice dell'ultimo elemento reso visibile con una chiamata a `makeVisible()`.

`Choice`

La classe `Choice` rappresenta il componente grafico generalmente conosciuto come `ComboBox`, così chiamato perché risultante dalla combinazione di un text field e una lista a selezione singola, che compare solo quando viene premuto un apposito pulsante a fianco del text field (lista a scomparsa). Una selezione sulla lista fa sì che l'elemento selezionato venga visualizzato nel text field. In diversi sistemi grafici il text field può essere reso editabile, consentendo all'utente di inserire anche stringhe non comprese tra gli elementi della lista. Questa modalità di funzionamento non è supportata in AWT: il text field è sempre a sola lettura e contiene sempre una copia dell'item selezionato.

I metodi del componente `Choice` sono quasi gli stessi della classe `List`, senza quelli relativi alla selezione multipla. Inoltre non esiste il metodo `deselect()`, che non è indispensabile in assenza di selezione multipla.

Tuttavia ci sono alcune differenze, che possono essere considerate vere e proprie incongruenze, dato che la natura di un oggetto `List` e quella della lista appartenente a un oggetto `Choice` è sostanzialmente la stessa. Si tratta di differenze poco intuibili che possono fuorviare chi usa le classi per la prima volta. In prima approssimazione si può dire che la `List` gestisce gli elementi prevalentemente attraverso i loro indici, mentre la `Choice` prevalentemente attraverso il loro valore.

Di conseguenza in `Choice` esistono alcuni metodi "orientati alle stringhe" che sono assenti in `List`:

```
select(String item)
```

che seleziona l'elemento uguale alla stringa data, e

```
String getItem(int index)
```

che restituisce l'item di dato indice.

Si tratta di metodi che sarebbero senz'altro utili anche nella classe `List`, e che riflettono una differenza di comportamento (tra l'altro non documentata nel `javadoc`) tra i due componenti nella gestione degli eventi: mentre per una `List` l'oggetto restituito dall'`ItemEvent` creato ad ogni selezione è un `Integer` che rappresenta l'indice dell'elemento selezionato o deselezionato, nel caso della `Choice` l'oggetto è una `String` che rappresenta direttamente l'elemento selezionato.

Un esempio sulle liste selezionabili

Il seguente esempio mostra il funzionamento di una `List` a selezione multipla e di una `Choice`. Nel `Form` compare una lista di colori disponibili tra i quali l'utente sceglie i colori da inserire in una palette. I colori selezionati (in una lista a selezione multipla) vengono inseriti nella `Choice`. L'utente può poi compiere un'ulteriore selezione tra i colori della palette. Il colore selezionato viene mostrato in un rettangolo accanto alla `Choice`.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ColorChoiceFrame extends Frame {
   Panel listPanel;
   Panel choicePanel;
   java.awt.List colorList;
   Choice selectedColorsChoice;
   FixedSizePanel colorPanel;
   Hashtable colors;

   public ColorChoiceFrame() {
      setBounds(100, 100, 300, 250);
      setTitle("Scelta colori");
      addWindowListener(new MainWindowListener());
      setLayout(new GridLayout(1, 2));
      createColorTable();
      add(createListPanel());
      add(createChoicePanel());
   }

   void createColorTable() {
      colors = new Hashtable();
      colors.put("Bianco", Color.white);
      colors.put("Grigio chiaro", Color.lightGray);
      colors.put("Grigio", Color.gray);
      colors.put("Grigio scuro", Color.darkGray);
      colors.put("Nero", Color.black);
      colors.put("Rosso", Color.red);
      colors.put("Rosa", Color.pink);
      colors.put("Arancio", Color.orange);
      colors.put("Giallo", Color.yellow);
      colors.put("Verde", Color.green);
      colors.put("Magenta", Color.magenta);
      colors.put("Cyan", Color.cyan);
      colors.put("Blu", Color.blue);
   }

   // gli altri metodi sono mostrati nel seguito
   ...
```

```
   public static void main(String[] args) {
      new ColorChoiceFrame().show();
   }
}
```

In questo esempio ci sono alcune variazioni nell'impostazione dei metodi `createXxx()`. Dato che il contenuto dei pannelli è un po' più complesso che negli esempi precedenti, il costruttore si limita a chiamare i metodi per la creazione dei `Panel`, e questi a loro volta creano direttamente i componenti più semplici da inserire al loro interno o chiamano a loro volta i metodi di creazione della `List` e della `Choice`. Per rendere il codice più omogeneo si è preferito questa volta non inserire i componenti creati dall'interno dei metodi di creazione, che invece restituiscono l'oggetto creato.

I "colori disponibili", corrispondenti a quelli predefiniti nella classe `Color`, vengono inseriti in una hashtable che consentirà di recuperare agevolmente sia i nomi che i corrispondenti oggetti `Color`.

Questi i metodi di creazione dei Panel:

```
Panel createListPanel() {
   listPanel = new Panel();
   Label listLabel = new Label("Colori disponibili", Label.CENTER);
   listLabel.setFont(new Font("Arial", Font.BOLD, 12));
   listPanel.add(listLabel);
   listPanel.add(createColorList());
   listPanel.add(new Label("Selezionare i colori"));
   listPanel.add(new Label("da inserire nella palette"));
   return listPanel;
}

Panel createChoicePanel() {
   choicePanel = new Panel();
   Label paletteLabel = new Label("          Palette          ",
                                 Label.CENTER);
   paletteLabel.setFont(new Font("Arial", Font.BOLD, 12));
   choicePanel.add(paletteLabel);
   colorPanel = new FixedSizePanel(new Dimension(110, 90));
   choicePanel.add(createSelectedColorsChoice());
   choicePanel.add(new Label("Selezionare il colore"));
   choicePanel.add(new Label("da visualizzare"));
   choicePanel.add(colorPanel);
   return choicePanel;
}
```

Questi metodi non fanno altro che inserire alcune `Label` e, rispettivamente, la `List` e la `Choice` all'interno dei `Panel`. In `createChoicePanel()`, si usa un espediente piut-

tosto rozzo per visualizzare la prima Label (`Palette`) con il `FlowLayout`, in modo che non compaia sulla stessa linea della successiva Label, inserendo una serie di spazi prima e dopo la stringa, in modo da rendere la Label sufficientemente lunga. Questo risulta necessario perché i parametri presi in considerazione dai layout per il calcolo delle dimensioni da assegnare (che saranno descritti nella sezione sui layout), nel caso della Label dipendono unicamente dal suo contenuto (le dimensioni assegnate con `setSize()` vengono ignorate dai layout). Per visualizzare il colore viene utilizzato un componente creato appositamente, il `FixedSizePanel`, descritto nella sezione *Layout*, sempre a causa della limitata controllabilità delle dimensioni dei componenti standard in presenza di layout.

Ed ecco i metodi più interessanti ai nostri fini, quelli che creano i due tipi di lista:

```
java.awt.List createColorList() {
   colorList = new java.awt.List(8, true);
   Enumeration colorNames = colors.keys();
   String item = null;
   while (colorNames.hasMoreElements()) {
      item = (String)colorNames.nextElement();
      colorList.add(item);
      if (item.equals("Bianco") || item.equals("Nero"))
         colorList.select(colorList.getItemCount() - 1);
   }
   colorList.addItemListener(
      new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            int index = ((Integer)e.getItem()).intValue();
            String item = colorList.getItem(index);
            if (e.getStateChange() == ItemEvent.SELECTED) {
               selectedColorsChoice.add(item);
               choicePanel.validate();
            }
            else
               selectedColorsChoice.remove(item);
         }
      }
   );
   return colorList;
}
```

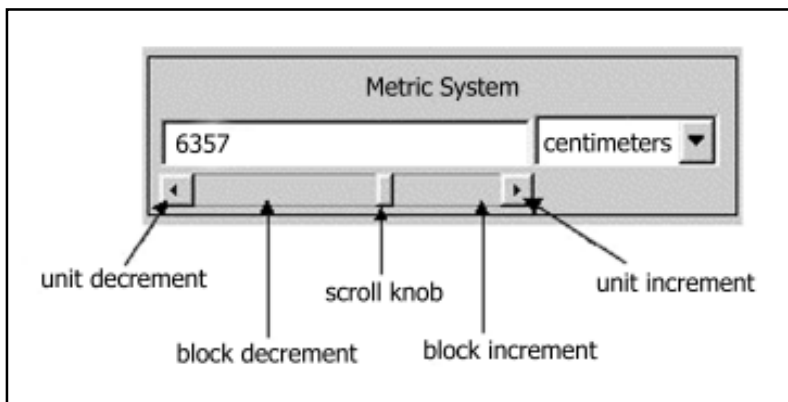
Inizialmente viene creata una `List` con 8 righe visibili, a selezione multipla (parametro true). Poi vengono inseriti nella lista i nomi dei colori precedentemente inseriti nella hashtable. I colori `Bianco` e `Nero` vengono selezionati. Il metodo del listener non fa altro che aggiungere o togliere l'elemento corrente alla `Choice`. Ma, a causa dell'infelice scelta di associare all'evento di selezione della `List` l'indice dell'elemento piuttosto che l'elemento stesso, il codice non risulta particolarmente pulito ed elegante. Quando viene ag-

giunto un elemento, viene chiamato il metodo `validate()` del `Panel`, per far sì che la `Choice` venga ridimensionata se necessario, ossia se la nuova stringa è più lunga del componente `Choice`.

```
Choice createSelectedColorsChoice() {
   selectedColorsChoice = new Choice();
   String[] selectedItems = colorList.getSelectedItems();
   for (int i = 0; i < selectedItems.length; i++)
      selectedColorsChoice.add(selectedItems[i]);
   colorPanel.setBackground((Color)colors.get(
      selectedColorsChoice.getItem(0)));
   selectedColorsChoice.addItemListener(
      new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            String item = (String)e.getItem();
            colorPanel.setBackground((Color)colors.get(item));
         }
      }
   );
   return selectedColorsChoice;
}
```

In quest'ultimo metodo, dopo la creazione la `Choice` viene inizializzata inserendo gli elementi preselezionati della lista, dopodiché il colore del `colorPanel` viene impostato opportunamente al colore del primo elemento inserito. Il colore viene prelevato dalla hashtable usando il nome come chiave.

Il metodo del listener è in questo caso molto più snello e immediatamente comprensibile, data la diversa impostazione della `Choice`, che associa all'evento di selezione diretta-

Figura 9.2 – *Una scrollbar con i suoi elementi costitutivi*

mente la stringa che rappresenta l'elemento selezionato. Una volta ottenuta tale stringa, questa viene di nuovo usata per recuperare il corrispondente oggetto `Color` dalla hashtable e impostare il colore di sfondo del `colorPanel`.

`ScrollBar`

Si è gia avuto avuto a che fare con questo componente, come elemento all'interno di altri componenti (`ScrollPane` e `TextArea`). Ora se ne esamineranno le caratteristiche nel dettaglio, e se ne esemplificherà l'uso come componente a sé stante. La figura 9.2 mostra le varie parti di cui è composta una Scrollbar:

La scrollbar ha un valore corrente variabile tra un minimo e un massimo (per default `1` e `100`). Tale valore viene rappresentato graficamente dalla posizione del suo elemento interno, lo *scroll knob* (traducibile approssimativamente con "manopola di scorrimento"). I due elementi sono interdipendenti: se cambia il valore cambia anche la posizione dello scroll knob, e viceversa. Il valore può essere modificato in diversi modi:

- spostando direttamente lo scroll knob con il mouse per trascinamento;

- cliccando sugli *unit increment buttons* che compaiono alle estremità del componente, incrementando o decrementando in questo modo il valore di una quantità "unitaria" (*unit increment*) definibile dall'utente (per default `1`);

- cliccando sull'area tra lo scroll knob e i pulsanti (*block increment area*). In tal modo si incrementa o decrementa il valore di una quantità, anch'essa definibile dall'utente (il default è 10) detta *block increment*;

- con la tastiera. Generalmente le frecce di spostamento e i tasti `PageUp` e `PageDown` determinano rispettivamente uno unit increment e un block increment.

La dimensione dello scroll knob è una frazione della lunghezza della block increment area uguale al rapporto tra il valore del *visible amount* e quello dellla differenza tra il valore massimo e il minimo. Il termine visible amount si riferisce al comune uso delle scrollbar per fare lo scrolling di un'area, come nello `ScrollPane`, in cui la dimensione dello scroll knob è proporzionale alla frazione dell'area che risulta visibile. Ad esempio se l'area totale ha un'altezza di 600 pixel e l'area visualizzata ha un'altezza di 100 pixel, lo scroll knob della scrollbar verticale avrà un'altezza pari a 1/6 di quella dell'intera area di scorrimento.

In realtà, poiché la scrollbar può essere usata anche in contesti differenti, ad esempio come uno slider (cioè un comando a scorrimento), il termine non risulta sempre appropriato; la proprietà indica in effetti la dimensione dello scroll knob, espresso in unità

uguali a `tot_len / (max - min)` , dove `max` e `min` indicano i valori massimo e minimo della Scrollbar e `tot_len` la lunghezza totale della block increment area (entro la quale si muove lo scroll knob).

Questi i principali metodi della classe `Scrollbar`:

Costruttori

```
Scrollbar()
```

Costruttore di default, costruisce una scrollbar verticale con valore minimo `0`, valore massimo `100`, valore iniziale `0`, unit increment `1`, block increment `10`, visible amount `10`.

```
Scrollbar(int orientation)
```

Costruisce una Scrollbar con l'orientamento dato; gli altri valori come il precedente.

```
Scrollbar(int orientation, int initialValue,int visibleAmount, int max, int min)
```

Costruisce una scrollbar con tutti i valori dati come argomenti tranne lo unit increment che rimane uguale a `1` e il block increment che rimane uguale a `10`.

Accesso ai parametri

I seguenti metodi restituiscono o modificano i parametri precedentemente descritti:

```
void setValues(int orientation, int value, int visibleAmount, int max, int min)
void setValue(int v)
int getValue()
void setOrientation(int v)
int getOrientation()
void setMinimum(int v)
int getMinimum()
void setMaximun(in tv)
int getMaximum()
void setUnitIncrement(int v)
int getUnitIncrement()
void setBlockIncrement(int v)
int getBlockIncrement()
```

```
void setVisibleAmount(int v)
int getVisibleAmount()
```

Quando vengono assegnati i valori, viene effettuato un controllo in modo da assicurare la reciproca coerenza dei vari parametri, in particolare che il minimo non sia superiore al massimo e che il valore sia compreso tra questi due estremi. In realtà il valore massimo che può essere assunto dal parametro `value` è pari a `max - visibleAmount`. Per capire questo comportamente decisamente poco intuitivo, bisogna considerare che la scrollbar è concepita principalmente per controllare la visualizzazione di un'area virtuale all'interno di una finestra, che viene "spostata" tramite la scrollbar stessa. In questa situazione, generalmente `max` rappresenta l'altezza totale (nel caso di scrolling verticale) dell'area virtuale, mentre `visibleAmount` rappresenta l'altezza dell'area di visualizzazione della finestra, ossia l'altezza della parte visibile; `value` rappresenta invece la coordinata `y`, rispetto all'origine dell'area virtuale, del punto visualizzato all'origine dell'area di visualizzazione, cioè al limite superiore dell'area stessa. Quando è visualizzata l'ultima parte dell'area virtuale (cioè quando lo scroll knob è "a fine corsa"), il limite inferiore dell'area virtuale e quello dell'area di visualizzazione coincidono. Di conseguenza il valore del parametro `value` (cioè la coordinata "virtuale" `y` della linea sul limite superiore dell'area di visualizzazione) corrisponderà all'altezza totale dell'area virtuale (`max`) meno l'altezza dell'area di visualizzazione.

Per il normale uso, ossia per il controllo di una scrolled area in genere si adopera uno `ScrollPane`, che si fa carico di controllare automaticamente le scrollbar. Invece le scrollbar sono spesso usate in AWT come slider (in mancanza di un componente apposito). Il prossimo esempio mostra questo tipo di utilizzazione.

Si tratta di un Frame che permette di definire i valori RGB di un colore tramite degli slider rappresentati da scrollbar, e mostra il colore risultante su un'area accanto agli slider. Ogni `Scrollbar` ha una `Label` e un `TextField` associati. La `Label` mostra il nome del colore base che la `Scrollbar` rappresenta e il `TextField` mostra il valore corrente. Inoltre il `TextField` fornisce un altro mezzo per modificare tale valore, inserendo direttamente un numero al suo interno. In tal caso lo stato della `Scrollbar` cambia di conseguenza. La `Scrollbar`, la `Label` e il `TextField` sono riuniti in una classe separata, chiamata `SliderPane`, che li incapsula in un componente riusabile in tutte le circostanze in cui serve uno slider. Questo rappresenta un esempio rudimentale di *custom component*, risultante dal semplice assemblaggio di altri componenti.

```
import java.awt.*;
import java.awt.event.*;

public class SliderPane extends Panel {
   Scrollbar scrollbar;
   Panel textPanel;
   TextField valueTextField;
   TextFieldListener valueTextFieldListener;

   public SliderPane(String label, int min, int max) {
      this(label, min, max, 0);
   }

   public SliderPane(String labelText, int min, int max, int initialValue) {
      setLayout(new GridLayout(2, 1));
      // l'ordine delle istruzioni seguenti deve essere mantenuto
      createScrollbar(Scrollbar.HORIZONTAL, min, max, initialValue);
      createTextPanel(labelText);
      add(textPanel);
      add(scrollbar);
   }

   void createTextPanel(String labelText) {
      textPanel = new Panel();
      textPanel.setLayout(new BorderLayout());
      textPanel.add(new Label(labelText, Label.CENTER), BorderLayout.CENTER);
      createValueTextField();
      textPanel.add(valueTextField, BorderLayout.EAST);
   }

   void createValueTextField() {
      int max = scrollbar.getMaximum();
      int numberOfDigits = Integer.toString(max).length();
      valueTextFieldListener = new TextFieldListener();
      valueTextField = new TextField(Integer.toString(scrollbar.getValue()),
                                                  numberOfDigits);
      valueTextField.addTextListener(
         new TextListener() {
            public void textValueChanged(TextEvent event) {
               String text = valueTextField.getText();
               try {
                  scrollbar.setValue(Integer.parseInt(text));
               } catch (Exception e) {
               }
            }
         }
      );
      valueTextField.addFocusListener (
         new FocusAdapter() {
```

```
            public void focusLost(FocusEvent e) {
               valueTextField.setText(Integer.toString(scrollbar.getValue()));
            }
         }
      );
   }

   Scrollbar createScrollbar(int orientation, int min, int max,
                             int initialValue) {
      int knobSize = (max - min) / 16;
      scrollbar = new Scrollbar(orientation, initialValue,
                                knobSize, min, max + knobSize);
      scrollbar.setBlockIncrement(knobSize);
      scrollbar.addAdjustmentListener (
         new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
               valueTextField.setText(Integer.toString(e.getValue()));
            }
         }
      );
      return scrollbar;
   }

   public int getValue() {
      return scrollbar.getValue();
   }

   public void setValue(int v) {
      scrollbar.setValue(v);
   }

   public void addListener(TextListener listener) {
      valueTextField.addTextListener(listener);
   }

   public void removeListeners(TextListener listener) {
      valueTextField.addTextListener(listener);
   }

   public void setSliderColor(Color c) {
      scrollbar.setBackground(c);
   }
}
```

Lo `SliderPane` è una sottoclasse di `Panel`, che contiene al suo interno una `Label`, un `TextField` e una `Scrollbar`. I primi due sono inseriti in un altro `Panel`, il `textPanel`, per poter essere visualizzati correttamente con i layout.

Nel costruttore viene prima creato il layout dello `sliderPane`, che è un `GridLayout` con due righe e una sola colonna. Le due righe saranno occupate dal `textPanel` e dalla `Scrollbar`, che quindi occuperanno spazi uguali. Quindi viene creata la scrollbar, utilizzando gli argomenti `min`, `max` e `initialValue`; per mantenere il codice abbastanza semplice, si è scelto di utilizzare soltanto delle scrollbar orizzontali, piuttosto che parametrizzare l'orientamento. L'argomento `labelText` viene passato al metodo `createTextPanel`, che sarà esaminato fra poco. In questo caso la creazione dei componenti e il loro inserimento nel `Frame` segue un ordine ben preciso. Infatti la scrollbar deve essere creata pre prima poché il suo valore viene utilizzato da `createTextPanel()` per inizializzare il `TextField`. Ma il `textPanel` deve essere inserito per primo perché deve comparire nella parte superiore del componente.

I metodi `createTextPanel()` e `createTextField()` creano in pannello superiore e il suo contenuto, ossia la `Label` e il `TextFiled`. Come layout si usa un `BorderLayout`, inserendo al centro la `Label`, e a destra (`BorderLayout.EAST`) il `TextField`, che viene creato con un numero di colonne (caratteri) uguale al numero di cifre del valore massimo. In tal modo la `Label` occupa tutta l'area del `Panel` lasciata libera dal `TextField`, con il testo posizionato al centro di tale area (allineamento `Label.CENTER`).

Al `TextField` viene assegnato un `TextListener` che ad ogni modifica del testo aggiorna il valore della scrollbar. Se il valore non è valido (fuori dal range oppure non numerico) viene generata una eccezione che viene intercettata senza compiere alcuna azione; in tal caso semplicemente il valore della scrollbar non viene modificato. Per assicurare che anche in questi casi il valore sia riallineato, viene aggiunto un `FocusListener` che assicura che il valore sia uguale a quello della `Scrollbar`, anche se solo allo spostamento del focus su un altro controllo.

Il metodo `createScrollbar()` crea una scrollbar con i valori iniziale, massimo e minimo passati come argomento. Il valore del visible amount è impostato a 1/16 del range totale, e lo stesso valore viene assegnato al block increment. Si noti che il valore massimo effettivo assegnato alla `Scrollbar` è uguale al valore massimo specificato più quello del visible amount, per i motivi precedentemente spiegati. Viene quindi assegnato alla `Scrollbar` un `AdjustmentListener` che modifica il testo del `TextField` in modo che il valore visualizzato sia sempre corrispondente a quello della `Scrollbar`.

Sulle interazioni bidirezionali tra componenti nella gestione degli eventi e sui problemi relativi, si veda la sezione dedicata agli eventi.

Vi sono infine alcuni metodi che permettono l'interazione con i client.

I metodi `setValue()` e `getValue()` danno accesso al valore della `Scrollbar`. I metodi `addListener()` e `removeListener()` permettono di aggiungere o rimuovere

TextListener esterni, che possano compiere azioni definite dal client ad ogni cambiamento del valore.

Il metodo setColor(), pur non indispensabile, permette di modificare il colore di sfondo della scrollbar, e si rivelerà utile per la classe RGBColorFrame.

Quello che segue è il codice del Frame che utilizza lo SliderPane.

```
import java.awt.*;
import java.awt.event.*;

public class RGBColorFrame extends Frame {
   SliderPane redSlider;
   SliderPane greenSlider;
   SliderPane blueSlider;
   Panel sliderPanel;
   Panel colorPanel;

   public RGBColorFrame() {
      setFrameProperties();
      add(createSliderPanel(), BorderLayout.CENTER);
      add(createColorPanel(), BorderLayout.EAST);
   }

   void setFrameProperties() {
      setTitle("Definizione colore");
      setBounds(100, 100, 250, 150);
      addWindowListener(new MainWindowListener());
   }

   Panel createSliderPanel() {
      sliderPanel = new Panel(new GridLayout(0, 1, 0, 4));
      createSliders();
      sliderPanel.add(redSlider);
      sliderPanel.add(greenSlider);
      sliderPanel.add(blueSlider);
      return sliderPanel;
   }

   void createSliders() {
      TextListener listener = new TextListener() {
         public void textValueChanged(TextEvent e) {
            setColor();
         }
      };

      redSlider = createSlider("Rosso", Color.red, listener);
      greenSlider = createSlider("Verde", Color.green, listener);
      blueSlider = createSlider("Blu", Color.blue, listener);
   }
```

```
   SliderPane createSlider(String label, Color color, TextListener listener) {
      SliderPane s = new SliderPane(label, 0, 255);
      s.setSliderColor(color);
      s.addListener(listener);
      return s;
   }

   Panel createColorPanel() {
      colorPanel = new Panel(new BorderLayout());
      colorLabel = new Label(" ");
      colorLabel.setFont(new Font("courier", Font.PLAIN, 50));
      colorPanel.add(colorLabel);
      setColor();
      return colorPanel;
   }

   void setColor() {
      Color color = new Color(redSlider.getValue(), greenSlider.getValue(),
                              blueSlider.getValue());
      colorLabel.setBackground(color);
   }

   public static void main(String[] args {
      new RGBColorFrame().show();
   }
}
```

La classe `RGBColorFrame` costruisce un `Frame` con all'interno due `Panel`: uno sistemato a sinistra, contenete a sua volta tre `SliderPane`, uno per ogni colore; uno a destra, contenente un `FixedSizePanel`, già comparso in un precedente esempio, usato anche questa volta per la visualizzazione del colore definito.

Il medoto `createSliderPanel()` crea un `Panel` che contiene tre `SliderPane`, sistemati con un `GridLayout`. Qui viene usato il costruttore `GridLayout(int rows, int cols, int hgap, int vgap)` in cui gli ultimi argomenti indicano la distanza orizzontale e verticale tre le celle. In questo caso viene assegnata una distanza verticale di 4 pixel.

Gli `SliderPane` vengono creati dal metodo `createSliders()` che a sua volta chiama `createSlider()`, che assegna ad ogni `SliderPane` il nome del colore, il colore stesso che sarà usato come colore di sfondo delle scrollbar e un listener che ad ogni modifica del valore chiama il metodo `setColor()`. Quest'ultimo imposta il colore della `colorLabel` secondo i valori RGB dei tre `SliderPane`.

`Canvas`

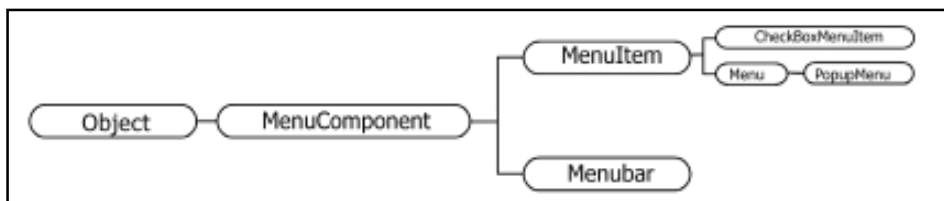
La classe `Canvas` è concepita come classe base per la creazione di componenti *custom*.

Se ne parlerà nella sezione dedicata a questo argomento.

Menu

`MenuComponent` e `MenuContainer`

I menu non fanno parte della gerarchia component, ma hanno una classe base separata, chiamata `MenuComponent`. La gerarchia è mostrata nella seguente fig. 9.3.

Figura 9.3 – *La gerarchia della classe* `MenuComponent`

Il motivo di ciò è che in molti sistemi grafici i componenti che fanno parte dei menu hanno molte meno features dei normali componenti. Infatti l'interfaccia della classe `MenuComponent` è molto ridotta rispetto a quella della classe `Component`. Escludendo i metodi relativi alla grafica e agli eventi, che saranno trattati nelle sezioni dedicate a questi argomenti, i seguenti sono i principali metodi di questa classe.

```
MenuContainer getParent()
```

A differenza del corrispondente metodo di `Component`, il `Parent` è un oggetto di una classe che implementa l'intefaccia `MenuContainer`, una semplice interfaccia con pochi metodi. La classe `Component` implementa questa interfaccia.

```
void setName(String s)
String getName()
```

Simili ai corrispondenti in `Component`.

`MenuBar`

La `MenuBar` rappresenta il classico contenitore dei menu, usato nella maggioranza dei

casi. Questi sono i principali compiti della `MenuBar` e i metodi relativi.

Per la gestione dei menu:

```
void add(Menu m)
```

Aggiunge un menu.

```
void remove(int index)
void remove(MenuComponent m)
```

Rimuovono un menu.

```
int getMenuCount()
```

Restituisce il numero di menu presenti nella `MenuBar`.

Per la gestione degli shortcuts (per il funzionamento degli shortcuts si veda più avanti la classe `MenuShortcut`):

```
Enumeration shortcuts()
```

Restituisce un elenco degli shortcuts presenti nella `MenuBar`.

```
void deleteShortcut(MenuShortcut s)
```

Cancella uno shortcut.

```
void getShortcutMenuItem(MenuShortcut s)
```

Restituisce il `MenuItem` associato con lo shortcut specificato.

Tra i componenti standard, la `MenuBar` è supportata soltanto dal `Frame`, con i metodi `setMenuBar()` e `getMenuBar()`.

MenuItem

La classe `MenuItem` rappresenta sia un item semplice, sia la classe base del componente `Menu`. Un item semplice è un componente dotato di una label che genera un

`ActionEvent` in risposta a un clic del mouse (o corrispondente evento di tastiera). Usato tipicamente per organizzare i comandi in maniera gerarchica in una serie di menu. Un tipo di item particolare è il *separator* che anziché un testo visualizza una linea di separazione. Queste i principali metodi:

```
MenuItem()
MenuItem(String label)
MenuItem(String label, MenuShortcut shortcut)
```

Stato

```
void setEnabled(boolean v)
boolean isEnabled()
```

Assegnano e restituiscono lo stato abilitato/disabilitato dell'item.

`Label`

```
void setLabel(String s)
String getLabel()
```

Per impostare e rilevare il valore dal testo visualizzato dall'item.

`Shortcut`

```
void setShortcut(MenuShortcut s)
```

Associa lo shortcut specificato all'item.

```
void deleteShortcut(MenuShortcut s)
```

Rimuove l'associazione dello shortcut specificato.

```
MenuShortcut getShortcut()
```

Restituisce lo shortcut associato.

`CheckboxMenuItem`

Questo componente si differenzia dal normale `MenuItem` per il fatto che assume uno

stato selezionato/non selezionato, allo stesso modo di un `Checkbox`, e che il click sul componente genera un `ItemEvent`. Un `CheckboxMenuItem` viene visualizzato con un simbolo a fianco quando è selezionato.

Lo stato può essere modificato o rilevato da codice con i metodi `void setState (boolean selected)` e `boolean getState()`.

Menu

Il componente `Menu` è un `MenuItem` utilizzato come contenitore di altri `MenuItem`. A tal fine implementa l'interfaccia `MenuContainer`. La classe `Menu` rappresenta un menu *pulldown* ossia a discesa, mentre la sottoclasse `PopupMenu` rappresenta i menu *popup*, usati soprattutto come *menu di contesto*.

Un menu può contenere semplici `MenuItem` oppure altri `Menu`, dando origine a una struttura ad albero a più livelli (menu gerarchico).

Ecco i principali metodi:

Costruttori

```
Menu()
```

```
Menu(String label)
```

Gestione degli item

```
void add(MenuItem item)
```

Aggiunge un item alla fine della lista.

```
void add(String s)
```

Crea un item con la label specificata.

```
void insert(MenuItem item, int index)
```

```
void insert(String label, int index)
```

Come i precedenti, ma inseriscono l'item nella posizione specificata.

```
void addSeparator()
```

`void insertSeparator(int index)`

Aggiungono o inseriscono un separatore.

`remove(MenuItem item)`

`void remove(int index)`

Rimuovono un item.

`void removeAll()`

Rimuove tutti gli item.

`int getItemCount()`

Restituisce il numero di item.

`MenuItem getItem(int index)`

Restituisce l'item che si trova nella posizione specificata.

PopupMenu

Questo componente ha la stessa interfaccia della classe base `Menu`, con in più un metodo `void show(Component origin, int x, int y)` che visualizza il menu alle coordinate specificate relative al componente specificato. Il `PopupMenu` deve essere contenuto, direttamente o indirettamente, nel componente specificato.

Il `PopupMenu` può anche essere usato come un normale `Menu`, ed essere inserito come tale in una `MenuBar`. In questo caso il metodo `show()` non può essere utilizzato.

Un `PopupMenu` può essere associato a qualunque componente tramite il metodo `void add(PopupMenu pm)` della classe `Component`.

MenuShortcut

Uno shortcut è un modo alternativo per attivare un `MenuItem` con un comando da tastiera. Lo shortcut è caratterizzato da un *keycode* che identifica un tasto della tastiera, e

un booleano che indica se il tasto *shift* deve essere premuto. I metodi essenziali sono:

```
int getKey()
```

Restituisce il keycode.

```
boolean usesShiftModifier()
```

Indica se lo shortcut prevede l'uso del tasto shift.

```
boolean equals(Shortcut s)
```

Confronta due shortcut.

Per associare uno shortcut a un `MenuItem` si usano gli appositi metodi della classe `MenuItem`.

Un esempio

Il seguente esempio mostra il funzionamento dei vari tipi di menu, compresi i popup. Il `Frame`, oltre ai menu, contiene soltanto una `TextArea` in cui vengono visualizzati messaggi generati da alcuni listener dei menu, e una `Label` che funge da "barra di stato" che viene mostrata o nascosta da un menu item.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class MenuDemo extends Frame {
   Hashtable colors;
   TextArea textArea = new TextArea();
   Label statusBar = new Label("Barra di stato");
   MenuBar menuBar = new MenuBar();
   MenuItem openFileItem;
   MenuItem closeFileItem;
   MenuItem saveFileItem;
   PopupMenu popupMenu;

   public MenuDemo() {
      setTitle("Menu Demo");
      setBounds(100, 100, 400, 300);
      addWindowListener(new MainWindowListener());
      setMenuBar(menuBar);
```

```
      colors = createColorTable();
      menuBar.add(createFileMenu());
      Menu windowMenu = createWindowMenu();
      menuBar.add(windowMenu);
      textArea = new TextArea();
      add(textArea, BorderLayout.CENTER);
      statusBar.setVisible(false);
      add(statusBar, BorderLayout.SOUTH);
      popupMenu = createPopupMenu();
      addPopupMenu(textArea);
      addPopupMenu(statusBar);
   }

   Hashtable createColorTable() {
      Hashtable colors = new Hashtable();
      colors.put("Bianco", Color.white);
      colors.put("Grigio", new Color(160, 160, 160));
      colors.put("Azzurro", new Color(110, 160, 255));
      colors.put("Giallo", new Color(255, 255, 160));
      colors.put("Verde", new Color(140, 255, 170));
      return colors;
   }
   ...
```

Nel costruttore vengono chiamati i metodi che creano i vari menu, e aggiunti alla `MenuBar`. Due chiamate al metodo `addPopupMenu()` collegano un `PopupMenu` alla `TextArea` e alla status bar. Il metodo `createColorTable` crea una `Hashtable` per collegare gli oggetti `color` alle stringhe dei menu item del sottomenu `color` del menu `window`.

```
   ...
   MouseAdapter mouseListener = new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
         int modifiers = e.getModifiers();
         if ((modifiers & InputEvent.SHIFT_MASK) != 0)
            popupMenu.show(component, e.getX(), e.getY());
      }
   };

   ActionListener closeListener = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         dispose();
      }
   }

   void addPopupMenu(Component c) {
      c.add(popupMenu);
```

```
        c.addMouseListener(mouseListener);
    }

    PopupMenu createPopupMenu() {
        PopupMenu menu = new PopupMenu();
        MenuItem closeItem = new MenuItem("Chiudi");
        closeItem.addActionListener(closeListener);
        menu.add(closeItem);
        return menu;
    }
```

Il popupMenu viene creato con `createPopupMenu()`, che inserisce nel menu un unico item a cui assegna il listener `closeListener` che chiude il `Frame`. Il metodo `addPopupMenu()` aggiunge al componente dato in argomento il popup menu e un `MouseListener` che apre il `PopupMenu` quando si preme un tasto del mouse insieme al tasto `SHIFT`. Notare che non è sufficiente aggiungere il menu al componente perché sia visualizzato, ma è necessario gestire gli eventi che determinano la sua apertura. Si è utilizzato un tasto del mouse più lo `SHIFT` piuttosto che il tasto destro, come si usa normalmente per i menu popup, perché spesso il tasto destro è gestito direttamente dal peer, quindi dal sistema, senza che sia possibile modificarne il funzionamento.

```
    ...
    Menu createFileMenu() {
        ActionListener fileMenuListener = new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                 MenuItem source = (MenuItem)e.getSource();
                 textArea.append("Ricevuto comando: " + source.getName() + '\n');
                 if (source
                          == openFileItem || source == closeFileItem) {
                    boolean enable = (source == openFileItem);
                    closeFileItem.setEnabled(enable);
                    saveFileItem.setEnabled(enable);
                 }
              }
           };

        Menu fileMenu = new Menu(«File»);
        openFileItem = createMenuItem(fileMenu, «Apri File»,
                                      «Apri», fileMenuListener);
        closeFileItem = createMenuItem(fileMenu, «Chiudi file»,
                                       «Chiudi», fileMenuListener);
        closeFileItem.setEnabled(false);
        fileMenu.addSeparator();
        saveFileItem = createMenuItem(fileMenu, «Salva file»,
                                      «Salva», fileMenuListener);
        saveFileItem.setEnabled(false);
```

```
        return fileMenu;
    }

    MenuItem createMenuItem(Menu menu, String name, String label,
                            ActionListener listener) {
        MenuItem item = new MenuItem(label);
        item.setName(name);
        item.addActionListener(listener);
        menu.add(item);
        return item;
    }
    ...
```

Il menu `File` contiene alcuni item per l'apertura, la chiusura e il salvataggio di un file. Tali operazioni non sono implementate, ma vengono invece generati messaggi che indicano quale item è stato selezionato. I menu item vengono abilitati e disabilitati coerentemente con la loro funzione (gli item `Salva` e `Chiudi` sono abilitati solo se c'è — virtualmente — un file aperto). I menu item sono generati con il metodo `createMenuItem`, che assegna al `MenuItem` una label, un nome e un listener.

```
    ...
    Menu createWindowMenu() {
        Menu windowMenu = new Menu("Window");
        windowMenu.add(createStatusBarItem());
        windowMenu.add(createColorMenu());
        return windowMenu;
    }

    MenuItem createStatusBarItem() {
        CheckboxMenuItem statusBarItem
        = new CheckboxMenuItem("Mostra barra di stato");
        statusBarItem.addItemListener (
            new ItemListener() {
                public void itemStateChanged(ItemEvent e) {
                    statusBar.setVisible(e.getStateChange() == ItemEvent.SELECTED);
                    validate();
                }
            }
        );
        return statusBarItem;
    }

    Menu createColorMenu() {
        ItemListener colorItemListener = new ItemListener() {
            CheckboxMenuItem currentItem;

            public void itemStateChanged(ItemEvent e) {
```

```
            if (currentItem != null)
               currentItem.setState(false);
            currentItem = (CheckboxMenuItem)e.getSource();
            currentItem.setState(true);
            String colorName = currentItem.getLabel();
            Color color = (Color)colors.get(currentItem.getLabel());
               textArea.setBackground((Color)colors.get(currentItem.getLabel()));
         }
      };

      Menu colorMenu = new Menu("Colore di sfondo");
      Enumeration colorNames = colors.keys();
      String colorName = null;
      CheckboxMenuItem item = null;
      while (colorNames.hasMoreElements()) {
         colorName = (String)colorNames.nextElement();
         item = new CheckboxMenuItem(colorName);
         item.addItemListener(colorItemListener);
         colorMenu.add(item);
         if (colorName.equals("Bianco"))
            colorItemListener.itemStateChanged(new ItemEvent(item,
                                                   ItemEvent.ITEM_STATE_CHANGED,
                                                   item, ItemEvent.SELECTED)
            );
      }
      return colorMenu;
   }

   public static void main(String[] args) {
      new MenuDemo().show();
   }
}
```

Il menu window contiene un item di tipo `CheckMenuItem` che visualizza o nasconde la status bar, e il sottomenu `color`. Quest'ultimo contiene dei `CheckMenuItem` usati per selezionare il colore di sfondo della `TextArea`. Soltanto un item alla volta può essere selezionato, come per i radio button. Per i menu questo comportamento non è automatico ma deve essere gestito dal listener, che infatti deseleziona l'item corrente prima di selezionare il nuovo.

I layout

Si è visto che in AWT in genere il posizionamento e/o il dimensionamento dei componenti all'interno di un `Container` sono affidati ai layout. Questa scelta è conforme a un "principio di economia" che suggerisce di evitare, per quanto possibile, ogni ripetizione

di codice promuovendo invece la riusabilità del codice stesso. Inserito nell'ambito della programmazione a oggetti, questo principio porta a progettare classi specializzate per compiere un determinato compito ogni qualvolta l'esecuzione di questo compito sia previsto in una pluralità di classi differenti e/o quando c'è la possibilità di svolgere il compito in maniere diverse, in alternativa fra loro. È appunto il caso dei layout. Tutti i componenti, a prescindere dalla loro natura, se si decide di usare un posizionamento/dimensionamento automatico, necessitano di metodi per calcolare le proprietà relative. Si potrebbe pensare di inserire i metodi direttamente nella classe base `Component`, ma in questo modo ci si vincolerebbe a un determinarto algoritmo. Affidando invece il compito a un altro oggetto si ha la possibilità di implementare una gerarchia di classi differenti, ciascuna delle quali usa un diverso algoritmo. Il layout funziona quindi come una sorta di *plug in* dell'oggetto `Container`, con cui si modifica il funzionamento di quest'ultimo.

Il layout dovrà anche conoscere l'interfaccia di `Component` per poterne modificare le proprietà.

`Layout`, `Container` e `Component` funzionano quindi in stretta collaborazione nel determinare le caratteristiche spaziali del componente, ma tutto questo avviene prevalentemente dietro le quinte, senza che il programmatore intervenga se non per assegnare il layout e, in alcuni casi, modificarne alcune proprietà. I componenti, infatti, non vengono assegnati direttamente al layout, ma invece al `Container` che a sua volta li assegnerà al layout.

Layout e constraint

I `Layout` in Java sono tali in quanto implementano l'interfaccia `LayoutManager`, che contiene i metodi base che consentono la collaborazione con i componenti. Esiste poi un'altra interfaccia, chiamata `LayoutManager2`, estensione di `LayoutManager`, che aggiunge una serie di metodi per la gestione dei cosiddetti *constraints*. Il concetto di constraint, traducibile con *vincolo*, si riferisce a qualunque condizione a cui sia assoggettato *uno specifico componente* nel momento in cui viene posizionato e dimensionato. Un esempio può essere la specifica zona del `BorderLayout` in cui viene inserito un componente (`CENTER`, `NORTH`, `EAST`, ecc.). In pratica il constraint viene specificato sotto forma di un generico `Object` (che nel caso del `BorderLayout` è una `String`) che viene passato come argomento del metodo `add` del `Container`; ad esempio:

```
someForm.add(someLabel, BorderLayout.NORTH);
```

Poiché il constraint è un generico `Object` (dal momento che il `Container` non può fare assunzioni sul tipo, dipendente dallo specifico layout), non si può contare su alcun controllo in fase di compilazione sull'argomento passato. Eventuali errori si manifesteranno solo in fase di esecuzione, e a volte in maniera non immediatamente individuabile. In

particolare se si passano constraint a layout che non ne usano, questi in genere vengono semplicemente ignorati. Per questo motivo è necessario fare particolare attenzione alla correttezza degli argomenti constraint che si passano al metodo `add`.

Tra i layout standard di AWT il `FlowLayout` e il `BorderLayout` implementano solo l'interfaccia `LayoutManager`, quindi non accettano constraint, mentre gli altri implementano `LayoutManager2` e quindi accettano constraint, come si vedrà fra poco.

I layout che non accettano constraint sono caratterizzati da un comportamento piuttosto rigido, dal momento che non c'è alcun controllo da parte del client (l'oggetto che utilizza la classe) sulle caratteristiche specifiche di layout del singolo componente.

Ciò non significa che un `LayoutManager` senza constraint non sia personalizzabile con l'assegnazione di opportuni parametri; ma si tratterà di parametri che avranno un effetto globale su tutti i componenti gestiti.

`LayoutManager`, `Component` e `Container`

I `LayoutManager` lavorano, come si è visto, in collaborazione con le classi `Container` e `Component`. Qui di seguito diamo una descrizione dei vari metodi che permettono questa collaborazione, per le classi `Component` e `Container`. I metodi delle interfacce `LayoutManager` e `LayoutManager2` hanno rilevanza solo per la creazione di custom layout, e non dovrebbero essere usati direttamente dal codice client del `Container`.

Classe `Component`:

```
float getAlignmentX()
float getAlignmentY()
```

Permettonno di specificare un allineamento orizzontale o verticale richiesto dal componente (e valutato dal `LayoutManager` a sua discrezione). Questi metodi sono utilizzati solo nelle classi `Swing` e non hanno effetto sui layout AWT.

```
Dimension getPreferredSize()
Dimension getMinimumSize()
Dimension getMaximumSize()
```

Questi metodi servono per dare informazioni al `LayoutManager` sulle dimensioni da assegnare al componente. Vedere il prossimo paragrafo per maggiori dettagli.

```
void doLayout()
void validate()
```

```
void invalidate()
```

Questi metodi, pur essendo parte dell'interfaccia di `Component`, in genere non hanno alcun effetto sui componenti che non siano `Container`, dal momento che solo il `Container` ha un reference al `LayoutManager` a cui in realtà vengono delegate le operazioni di layout. È quindi da evitare l'uso di questi metodi su componenti non derivati da `Container`. Per questo motivo la descrizione dei metodi suddetti viene riferita qui sotto alla classe `Container`.
Classe `Container`:

```
void add(Component c)
void add(Component c, int index),
void add(Component c, Object constraint)
void add(Component c, Object constraint, int index)
```

Aggiungono un componente (eventualmente nella posizione specificata) al `Container`; il componente viene anche aggiunto al `LayoutManager`, passando eventualmente il constraint.

```
void setLayout(LayoutManager lm)
```

Assegna un `LayoutManager` al `Container`.

```
LayoutManager getLayout()
```

Restituisce un reference al `LayoutManager`.

```
void doLayout()
```

Fa sì che venga ricalcolato il layout dei componenti all'interno del `Container`; l'operazione viene delegata al `LayoutManager`. Questo metodo in generale non dovrebbe essere chiamato dal client, il quale dovrebbe invece usare `validate()`.

```
void validate()
```

Determina il ricalcolo del layout all'interno del `Container` in maniera ricorsiva (ogni

componente che sia a sua volta un `Container` ricalcola il layout) per tutti i componenti che risultano non validi; un componente può essere invalidato esplicitamente dal client con il metodo `invalidate()` oppure indirettamente da altri metodi che aggiornano proprietà che influiscono sul layout. Se ci sono variazioni nel layout questa chiamata determina un ridisegno del componente sullo schermo.

```
void invalidate()
```

Marca il componente come non più valido, il che significa che necessita di un ricalcolo del layout. L'operazione sarà compiuta alla successiva esecuzione del metodo `validate()`.

```
void getInsets(Insets i)
Insets getInsets()
```

Metodi di accesso all'oggetto `Insets` associato al `Container`. La classe `Insets`, che contiene i quattro campi pubblici `left`, `right`, `top`, `bottom`, rappresenta le dimensioni del bordo del `Container`. Questi valori vengono utilizzati dal `LayoutManager` per calcolare posizione e dimensioni dei componenti.

Dimensionamento

Si è detto che il compito del `LayoutManager` non è limitato alla disposizione dei componenti all'interno del container, ma include anche il dimensionamento. Questo significa che un componente, se affidato al `Layout`, non ha più il controllo non solo della sua posizione, ma neppure delle sue dimensioni. In pratica i metodi `setLocation()`, `setSize()` e `setBounds()` non avranno effetto. Come vengono stabilite quindi le dimensioni di un componente? In realtà dire che il componente non ha il controllo dei propri parametri spaziali non è del tutto esatto. Il `LayoutManager`, nel calcolare le dimensioni, si basa su quanto ricevuto dal metodo `getPreferredSize()` della classe `Component`, che ogni componente implementa per conto suo. Tuttavia, come dice il nome, questa dimensione è solo la dimensione *preferita* del componente, quindi *non è vincolante* per il `LayoutManager`. In generale le operazioni di layout si basano su una serie di *richieste* da parte del componente che vengono prese in considerazione dal `LayoutManager` insieme ad altri fattori. Dalla combinazione di tutto ciò risultano la dimensione e la posizione assegnate, secondo lo specifico algoritmo di ciascun `LayoutManager`. Altri metodi presi in considerazione dai layout (ma non sempre da tutti) sono `getMinimumSize()` e `getMaximumSize()` che pongono dei limiti alle dimensioni assegnabili, nei casi in cui la dimensione assegnata non sia quella preferita.

Purtroppo nella classe `Component` non esistono i corrispondenti metodi `set` dei tre metodi `get` appena menzionati (questo è stato corretto nel package `Swing`, in cui la classe `JComponent` include invece questi metodi nella sua interfaccia). Ciò rende abbastanza scomodo assegnare queste proprietà ai componenti: in pratica è necessario creare delle sottoclassi dei componenti e ridefinire il metodo `getPreferredSize()` ed eventualmente gli altri due.

Un esempio è dato dalla classe `FixedSizePanel`, utilizzata in alcuni precedenti esempi per la visualizzazione di un colore; in quei casi serviva un `Panel`, usato semplicemente per visualizzare un certo colore di background, ma che avesse dimensioni assegnabili a piacere (sia pure con la riserva che non c'è la garanzia che tutti i `LayoutManager` rispettino le dimensioni preferite), piuttosto che dipendenti dal contenuto (in questo caso nullo).

Il seguente è il codice che definisce la classe.

```
import java.awt.*;

public class FixedSizePanel extends Panel {
   public Dimension getPreferredSize() {
      return getSize();
   }
}
```

Si tratta di un discendente di `Panel` che ridefinisce il metodo `getPreferredSize()` della classe base.

Il metodo della classe base restituisce un valore che dipende unicamente dal contenuto, cosicché un `Panel` vuoto avrà dimensioni nulle o minime.

Il metodo ridefinito invece restituisce il valore di `getSize()`, cioè le dimensioni normalmente usate in assenza di layout, assegnabili per mezzo del metodo `setSize()`.

Un altro elemento usato dai `LayoutManager` per il calcolo delle dimensioni è l'oggetto `Insets` associato al `Container`, che definisce le dimensioni dei bordi, cioè i margini che dovrebbero essere lasciati liberi sui lati del `Container`. Come nel caso delle dimensioni, non esiste un metodo `setInsets()` nella classe `Container` e per assegnare esplicitamente dei margini è necessario creare una sottoclasse apposita e ridefinire il metodo `getInsets()`.

In generale, i `LayoutManager` determinano posizione e dimensioni assegnando una porzione di spazio ad ogni componente, calcolata tenendo conto dello spazio disponibile nel `Container` e delle dimensioni preferite, minime e massime assegnabili al componente. A seconda dei casi (ossia a seconda dell'algoritmo usato ed eventualmente del valore di certi parametri), il componente potrà essere "stirato" per occupare l'intera area a sua disposizione, e in questo caso si adatterà ad ogni modifica della dimensione del `Container`, e quindi delle aree destinate a ciascun componente, oppure conserverà una sua dimensione propria e sarà lo spazio attorno (il margine della sua area) a variare ad ogni modifica.

Il metodo `pack()`

Negli esempi visti finora, la finestra principale è stata sempre dimensionata esplicitamente usando il metodo `setSize()` oppure `setBounds()`. Esiste un'altra possibilità: con il metodo `pack()` la finestra assume le sue dimensioni "ottimali" in base alle dimensioni preferite dei componenti in essa contenuti, e questi vengono disposti, compatibilmente con il proprio layout, in modo da occupare il minor spazio possibile. Se la finestra non è visibile, viene visualizzata. Il metodo `pack()` è dichiarato nella classe `Window`, quindi è utilizzabile da tutte le sue sottoclassi.

`FlowLayout`

Il `FlowLayout` è il più semplice (almeno concettualmente) tra i `LayoutManager` standard. Dispone i componenti uno dopo l'altro con le stesse modalità delle parole di un documento in un word processor: da sinistra a destra nell'ordine di inclusione, quindi nello spazio sottostante (andando "a capo") sempre da sinistra a destra. Gli unici parametri che possono essere assegnati al `Layout` (in genere nel costruttore, ma anche con appositi metodi) sono *l'allineamento* (alignment) orizzontale e il *gap* orizzontale e verticale. L'allineamento funziona come per il testo (si assegna una delle costanti `LEFT`, `CENTER`, `RIGHT`); il default è `CENTER`. Il gap è la distanza (orizzontale o verticale) tra i componenti e dal bordo del `Container`; il default è 5 unità (pixel).

Il `FlowLayout` è il layout di default di tutti i `Panel`.

`GridLayout`

Il `GridLayout` è, come il `FlowLayout`, privo di gestione di constraint; i componenti vengono disposti in forma di griglia, in celle disposte su righe e colonne, ma tutte le celle sono di uguale dimensione, indipendentemente dal contenuto. La dimensione delle celle sarà tale da far sì che le celle occupino l'intero spazio del `Container`. Il numero di righe e colonne può essere prefissato, oppure può essere determinato automaticamente sulla base del numero dei componenti inseriti: se il numero di righe è `0`, ossia indefinito, il numero delle colonne deve essere maggiore di `0`, e viceversa. Quando il valore del numero di righe o colonne è `0`, esso sarà stabilito in base al numero dei componenti inseriti. Oltre al numero di righe e colonne, si possono specificare un gap orizzontale e uno verticale, come per il `FlowLayout`. I parametri anche in questo caso possono essere passati al costruttore o assegnati con metodi specifici.

`BorderLayout`

Il `BorderLayout` suddivide l'area del `Container` in 5 zone corrispondenti al centro e ai lati dell'area rettangolare. Ne consegue che in un BorderLayout si possono inserire al

massimo 5 componenti (naturalmente ognuno di essi può essere a sua volta un `Container`). La zona in cui inserire il componente viene specificata come constraint, passando una delle costanti `CENTER`, `NORTH`, `SOUTH`, `EAST`, `WEST`. Le dimensioni di ciascuna zona dipendono dalle dimensioni del container e dalle dimensioni preferite del componente. Ogni zona ha dimensioni uguali a quelle del componente, più i valori del gap orizzontale e verticale.

Per essere più precisi, valgono le seguenti regole:

- Nelle zone `NORTH` e `SOUTH`, la dimensione orizzontale del componente inserito sarà adattata a quella della zona, a sua volta uguale a quella del `Container`. La dimensione verticale sarà quella preferita del componente.

- Nelle zone `WEST` ed `EAST` la dimensione verticale del componente sarà uguale a qualla della zona, che a sua volta è uguale alla dimensione verticale del `Container` meno le dimensioni verticali delle zone `NORTH` e `SOUTH`. La dimensione orizzontale sarà quella preferita del componente.

- Nella zona `CENTER` le dimensioni orizzontale e verticale del componente saranno adattate entrambe a quelle della zona, che occupa tutto lo spazio lasciato libero dalle altre zone.

- La zona di default è `CENTER`. Ogni zona può essere lasciata vuota, senza che le suddette regole cambino.

`CardLayout`

Il `CardLayout` tratta i componenti inseriti come se si trattasse di schede (card) disposte in una pila lungo un immaginario asse z perpendicolare alla superficie di visualizzazione. Le schede (ossia i componenti) sono visibili uno alla volta e possono essere selezionati andando avanti e indietro sulla lista oppure per nome.

Il nome della scheda deve essere specificato come constraint nel metodo `add()` del container. È consigliabile usare come nome lo stesso nome del componente.

Le schede vengono selezionate con i seguenti metodi della classe `CardLayout`:

```
void next(Container c)
```

Rende visibile la scheda successiva a quella corrente.

```
void previous(Container c)
```

Rende visibile la scheda precedente quella corrente.

```
void first(Container c)
```

Rende visibile la prima scheda.

```
void last(Container c)
```

Rende visibile l'ultima scheda.

```
void show(Container c, String cardName)
```

Rende visibile la scheda di dato nome.

Come si vede, tutti i metodi prendono come argomento un reference al `Container`; infatti un `LayoutManager` non è legato a un singolo `Container`, ma potrebbe essere condiviso da più `Container`.

Il seguente esempio mostra l'uso dei `LayoutManager` descritti finora.

```
import java.awt.*;
import java.awt.event.*;

public class LayoutDemo extends Frame {
   public static final String FIRST = "First";
   public static final String LAST = "Last";
   public static final String NEXT = "Next";
   public static final String PREVIOUS = "Previous";

   Panel cardContainer;
   CardLayout cardLayout = new CardLayout();
   Choice cardChoice;

   public LayoutDemo() {
      setTitle("Layout Demo");
      setBounds(100, 100, 640, 400);
      addWindowListener(new MainWindowListener());
      cardChoice = createCardChoice();
      cardContainer = createCardContainer();
      add(cardChoice, BorderLayout.NORTH);
      add(cardContainer, BorderLayout.CENTER);
      add(createNavigatorPanel(), BorderLayout.SOUTH);
   }
```

```
   ...

   public static void main(String[] args) {
      new LayoutDemo().show();
   }
}
```

Il `Frame` usa il `BorderLayout` di default. Nella parte centrale viene inserito un `Panel` che utilizza un `CardLayout`. In quella superiore (`NORTH`) si trova un `Choice` usato per selezionare la scheda per nome; in quella inferiore (`SOUTH`) un `Panel` contenente quattro "pulsanti di navigazione" che visualizzano rispettivamente la prima scheda, la precedente, la successiva, l'ultima. Le quattro costanti statiche definite all'inizio serviranno per identificare questi pulsanti.

```
   ...

   public Panel createCardContainer() {
      Panel p = new Panel();
      p.setLayout(cardLayout);
      addCard(p, "Card 1", "FlowLayout con hgap=0 e vgap=0",
               new FlowLayout(FlowLayout.CENTER, 0, 0));
      addCard(p, "Card 2", "FlowLayout con allineamento a sinistra,
               hgap=10 e vgap=20", new FlowLayout(FlowLayout.LEFT, 10, 20));
      addCard(p, "Card 3", "GridLayout con 2 colonne", new GridLayout(0, 2));
      addCard(p, "Card 4", "GridLayout con 2 righe", new GridLayout(2, 0));
      addCard(p, "Card 5", "GridLayout con 3 righe e 3 colonne",
               new GridLayout(3, 3));
      return p;
   }

   void addCard(Container container, String cardName,
                String labelText, LayoutManager layout) {
      Panel p = new Panel();
      p.setName(cardName);
      p.setLayout(layout);
      Label l = new Label(labelText, Label.CENTER);
      l.setFont(new Font("Arial", Font.BOLD, 12));
      p.add(l);
      p.add(new TextArea());
      p.add(new Checkbox("Checkbox Uno"));
      p.add(new Checkbox("Checkbox Due"));
      p.add(new Checkbox("Checkbox Tre"));
      p.add(new Button("Button Uno"));
      p.add(new Button("Button Due"));
      container.add(p, cardName);
      cardChoice.addItem(cardName);
   }
   ...
```

Le diverse schede sono rappresentate da 5 componenti `Panel` che hanno al loro interno componenti uguali ma sistemati con `LayoutManager` differenti. Per creare un `Panel` si usa il metodo `addCard()` a cui si passa il `Container`, il nome della scheda, una descrizione che viene visualizzata con una `Label`, e il `LayoutManager`. Poiché per costruire i diversi `Panel` viene usato sempre lo stesso metodo, non si possono usare layout che prevedono l'uso di constraint, che cambierebbero a seconda del layout; quindi si usano solo `FlowLayout` e `GridLayout` con configurazioni interne differenti. Le caratteristiche dei layout sono mostrate nelle stringhe di descrizione. Gli esempi di `GridLayout` comprendono colonne fisse e righe indefinite, righe fisse e colonne indefinite, righe e colonne fisse. Ad ogni `Panel` viene assegnato come nome lo stesso nome usato per identificare il componente all'interno del `CardLayout`. Alla fine del metodo `addCard()`, il nome della sceda viene aggiunto al componente `Choice` creato nel metodo seguente, in cui gli viene assegnato un `ItemListener` che visualizza la scheda con il nome corrispondente all'elemento selezionato.

```
...
      public Choice createCardChoice() {
   final Choice choice = new Choice();

   ItemListener listener = new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
         cardLayout.show(cardContainer, choice.getSelectedItem());
      }
   };

   choice.addItemListener(listener);
   choice.setFont(new Font("Arial", Font.BOLD, 20));
   return choice;
}

public Panel createNavigatorPanel() {
   ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         int index = cardChoice.getSelectedIndex();
         String name = ((Component)e.getSource()).getName();

         if (name.equals(FIRST)) {
            cardLayout.first(cardContainer);
            index = 0;
         }
         else if (name.equals(LAST)) {
            cardLayout.last(cardContainer);
            index = cardContainer.getComponentCount() - 1;
         }
         else if (name.equals(NEXT)) {
```

```
            cardLayout.next(cardContainer);
            index++;
         }
         else if (name.equals(PREVIOUS)) {
            cardLayout.previous(cardContainer);
            index—;
         }

         cardChoice.select(index);
      }
   };

Panel p = new Panel();
p.setLayout(new GridLayout(1, 0));
p.add(createNavigatorButton(FIRST, "|<", listener));
p.add(createNavigatorButton(PREVIOUS, "<", listener));
p.add(createNavigatorButton(NEXT, ">", listener));
p.add(createNavigatorButton(LAST, ">|", listener));
return p;
}

Button createNavigatorButton(String name, String label,
                              ActionListener listener) {
   Button b = new Button(label);
   b.setName(name);
   b.setFont(new Font("Courier", Font.BOLD, 20));
   b.addActionListener(listener);
   return b;
}

...
```

Nel metodo `createNavigatorPanel()` viene definito un `ActionListener` che verrà assegnato a tutti i pulsanti di navigazione. Il listener seleziona la funzione sulla base del nome del pulsante, corrispondente a una delle costanti precedentemente definite (`FIRST`, `PREVIOUS`, `NEXT` o `LAST`). Poiché il `CardLayout` non ha nessun metodo che restituisca il componente corrente, per poter aggiornare la `Choice` ad ogni modifica causata dai pulsanti, occorre individuare l'indice corrente con un altro sistema: viene definita una variabile `index` che viene inizializzata all'indice corrente della `Choice` (corrispondente all'indice della scheda visualizzata); poi, a seconda dei casi l'indice viene impostato a `0` (`FIRST`), all'indice dell'ultimo elemento (`LAST`) oppure decrementato (`PREVIOUS`) o incrementato (`NEXT`). Infine l'indice così modificato viene usato per selezionare il nuovo elemento sulla `Choice`.

Il `Panel` creato da questo metodo usa un `GridLayout` con una sola riga in modo che i pulsanti vengano visualizzati uno a fianco all'altro e siano di dimensioni uguali. Ad ogni pulsante viene assegnato il nome corrispondente alla sua funzione.

GridBagLayout

Il `GridBagLayout` è il più complesso ma anche il più flessibile tra i `LayoutManger` standard AWT. Il principio di funzionamento è relativamente semplice: l'area del container viene suddivisa in una serie di celle, come nel `GridLayout`, ma a differenza di questo le righe e le colonne possono avere ciascuna dimensioni differenti. Inoltre ciascun componente può occupare una o più celle; lo spazio occupato da un componente è detto *area di visualizzazione*. Il componente può essere visualizzato con varie modalità all'interno della sua area di visualizzazione, ad esempio specificando un margine, una modalità di allineamento, ecc. Tutti i parametri relativi alla visualizzazione del singolo componente vengono specificati come constraint per mezzo di un oggetto della classe `GridBagConstraints`.

La classe `GridBagConstraints` contiene una serie di campi *pubblici*, i cui valori sono quindi liberamente leggibili o assegnabili da qualunque oggetto. In questo caso non viene osservata la regola di non esporre mai i dati della classe, dato che l'unica funzione di `GridBagConstraints` è di raggruppare una serie di parametri utilizzati dal `LayoutManager`. Ecco un elenco dei parametri in questione con il rispettivo significato:

`gridx`

`gridy`

Specificano l'indice della colonna (`gridx`) e della riga (`gridy`) che rappresentano l'"origine" del componente, ossia indicano la cella in cui si trova l'angolo superiore sinistro del componente. I valori degli indici partono da `0` (la cella in alto a sinistra). Il valore di default è quello di una speciale costante, `GridBagConstraints.RELATIVE`. Questo valore serve per indicare che il componente deve essere posizionato sulla destra (per `gridx`) o sotto (per `gridy`) l'ultimo componente precedentemente inserito.

`gridwidth`

`gridheight`

Specificano il numero di colonne (`gridwidth`) o righe (`gridheight`) occupate dall'area di visualizzazione. Il valore di default è `1`. Se si usa come valore la costante `GridBagConstraints.REMAINDER` il componente occuperà tutte le restanti colonne (per `gridwidth`) o righe (per `gridheight`).

`fill`

I valori di questo parametro indicano se e come la dimensione del componente deve essere ampliata per adattarsi alle dimensioni dell'area di visualizzazione, se questa è più

grande del componente. I valori che si possono assegnare sono `GridBagConstraints.NONE` (il default; il componente conserva le sue dimensioni), `GridBagConstraints.HORIZONTAL` (il componente occuperà tutta la larghezza dell'area di visualizzazione), `GridBagConstraints.VERTICAL` (il componente occuperà tutta l'altezza dell'area di visualizzazione), e `GridBagConstraints.BOTH` (il componente occuperà l'intera area di visualizzazione).

`ipadx`

`ipady`

Specificano le dimensioni in pixel del margine interno del componente. Il valore di default è `0`. Le dimensioni minime del componente saranno date da quelle restituite da `getMinimumSize()` più `ipadx*2` per la larghezza e più `ipady*2` per la lunghezza.

`insets`

Specifica le dimensioni del margine esterno del componente. Il valore di default è `0`. Il margine esterno è la distanza minima (orizzontale e verticale) tra i bordi del componente e quelli dell'area di visualizzazione. I valori sono specificati da un oggetto della classe `Insets`.

`anchor`

Indica come il componente è "ancorato" all'area di visualizzazione; in sostanza definisce l'allineamento orizzontale e verticale. I valori assegnabili sono `GridBagConstraints.CENTER` (il valore di default), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST`, `GridBagConstraints.SOUTHEAST`, `GridBagConstraints.SOUTH`, `GridBagConstraints.SOUTHWEST`, `GridBagConstraints.WEST` e `GridBagConstraints.NORTHWEST`. Se i valori sono `NORTH` o `SOUTH`, l'allineamento orizzontale sarà al centro; se i valori sono `EAST` o `WEST`, l'allineamento verticale sarà al centro.

`weightx`

`weighty`

Specifica di quanto una riga o una colonna deve cambiare di dimensione se il container

viene ridimensionato. Il default è `0` per tutte le righe e le colonne. In questo caso le dimensioni delle celle non cambiano se il container viene ridimensionato e nel caso di aumento delle dimensioni, occuperanno lo spazio al centro del container. Se una o più righe (colonne) hanno un valore maggiore di `0` la sua dimensione varierà al ridimensionamento del container. Se la dimensione viene aumentata, la quantità di spazio aggiunta sarà distribuito tra tutte le righe o colonne che hanno `weight` maggiore di `0`, in proporzione al loro valore.

Il calcolo viene fatto sulla base della somma dei valori di `weightx` e `weighty`: ad ogni riga o colonna sarà assegnato uno spazio extra corrispondente alla frazione del totale (ad esempio: `weightx / totalWeightx`).

Se ad esempio in un `GridBagLayout` ci sono quattro colonne con `weightx` uguali a `3`, `2`, `0`, `5`, e la dimensione del container viene aumentata di 50 pixel, la prima colonna sarà allargata di 3 / 10 * 50 = 15 pixel, la seconda di 2 / 10 * 50 = 10 pixel, la terza conserverà la sua dimensione, la quarta sarà allargata di 5 / 10 * 50 = 25 pixel.

Questo parametro si riferisce a righe e colonne, ma in realtà è di solito specificato come parametro dell'oggetto `GridBagConstraint` (può essere anche assegnato direttamente al `LayoutManager`, si veda sotto), quindi riferito al componente. Il valore effettivo, nel caso di più valori assegnati a componenti disposti sulle stesse righe o colonne, è il massimo tra quelli dei diversi componenti.

Alcuni parametri di layout possono essere modificati anche agendo direttamente sull'oggetto `GridBagLayout`. Qui sotto si riporta un elenco dei campi e dei metodi più frequentemente utilizzati:

```
int[] columnWidths

int[] rowHeights[]

double[] columnWeight

double[] rowWeight
```

Questi campi pubblici permettono di assegnare delle dimensioni e dei parametri `weight` prefissati alle colonne e alle righe. Se i valori sono in numero superiore alle righe o alle colonne, i valori in eccesso saranno ignorati; se sono in numero inferiore, le ultime righe o colonne avranno come dimensione quello dell'ultimo elemento dell'array. Se questi valori non vengono assegnati, le dimensioni saranno calcolate automaticamente dal `LayoutManager`.

```
void setConstraints(Component comp, GridBagConstraints constr)
```

Con questo metodo è possibile modificare i constraints di un componente dopo il suo inserimento nel container.

Custom layout

In AWT è possibile definire dei custom layout che implementano particolari algoritmi di layout definiti dall'utente. Per realizzare un `custom LayoutManager` si deve definre una classe che implementi l'interfaccia `LayoutManager` (eventualmente `LayoutManager2`). La trattazione della creazione di un `LayoutManager` custom esula dagli scopi del presente capitolo.

Gli eventi

Il modello gerarchico in Java 1.0

Il modello degli eventi in Java 1.0 è notevolmente diverso da quello di Java 1.1. Verrà descritto brevemente per completezza e perché esiste comunque del codice in circolazione scritto secondo questo modello. Se ne sconsiglia decisamente l'adozione in tutto il nuovo codice: il modello gerarchico è stato sostituito perché poco funzionale e poco conforme alle regole di un buon design object oriented.

Nel modello gerarchico, l'evento viene gestito direttamente dal componente da cui trae origine, per mezzo del metodo `action()`, usato per gli eventi corrispondenti agli `ActionEvent` di Java 1.1, o `handleEvent()` per qualunque evento. Se l'evento non viene gestito dal componente, viene passato al suo container (sempre al metodo `action()` o `handleEvent()`), e così via fino ad arrivare alla finestra principale. I diversi eventi si distinguono tra loro per un codice di identificazione (ID).

Questo sistema ha diversi svantaggi: per gestire un evento è necessario definire una sottoclasse di un componente in cui si ridefinisce il metodo `action()` o `handleEvent()`; in `handleEvent`, tutti gli eventi sono gestiti da un unico metodo, per cui si è costretti ad usare molto codice condizionale per distinguere i vari tipi di evento; l'evento può essere gestito solo dal componente interessato o da un suo container e non da un oggetto specializzato; l'evento viene passato ai vari metodi `handleEvent` anche se non viene affatto gestito, consumando inutilmente risorse; gli eventi sono organizzati, in contrasto con la logica object oriented, come oggetti di un'unica classe e si differenziano solo per le loro proprietà. La natura degli eventi è invece tale da rendere molto più appropriato definire una classe per ogni diverso tipo di evento.

Il delegation model in Java 1.1

La versione di AWT contenuta in Java 1.1 segna un grande passo in avanti per quanto riguarda la gestione degli eventi: da un sistema rozzo e farraginoso si passa al modello forse più avanzato tra quelli attualmente esistenti nelle librerie GUI di qualunque linguaggio. Viene adottato il cosiddetto *delegation model* che utilizza un noto design pattern, l'*observer*: gli eventi vengono "osservati", o meglio "ascoltati" da oggetti che implementano determinate interfacce, ognuna delle quali è specializzata per un determinato tipo di

evento. Gli eventi sono ora distinti per tipo e sono implementati con una gerarchia di classi, non più con una singola classe. Il fatto che i gestori degli eventi, detti *listener*, siano definiti come interfacce toglie ogni restrizione al modo in cui possono essere implementati: una classe può implementare un singolo listener o più listener, la funzione di listener può essere svolta dallo stesso componente o da una classe specializzata. Inoltre, il fatto che il listener sia slegato dal componente che origina l'evento permette a diversi componenti di condividere lo stesso listener, come è anche possibile che uno stesso componente abbia diversi listener o addirittura che uno stesso evento sia gestito da più listener; tutto con la massima flessibilità possibile.

Il funzionamento, già descritto ed esemplificato in precedenza, viene qui brevemente riassunto: per ogni evento che si vuole gestire viene implementato un listener, che implementa l'interfaccia corrispondente al tipo di evento che si vuole gestire. Il listener può essere implementato in diversi modi: in una classe che ha già altre funzionalità, oppure in una classe specializzata con la stessa classe che può implementare uno o più listener. Il listener viene aggiunto al componente che genera l'evento per mezzo di un metodo `addXxxListener`, dove `Xxx` rappresenta il tipo di evento, considerando che a un componente può essere assegnato un numero arbitrario di listener; se un listener viene assegnato a più componenti, e deve distinguere l'oggetto che ha generato l'evento, può usare il metodo `getSource()` dell'evento e in questo caso sarà presente codice condizionale. È sempre possibile e generalmente consigliabile ricorrere a listener diversi per ciascun componente.

La flessibilità del modello è sicuramente un pregio, ma lascia anche ampie possibilità di usare il modello in maniera impropria, e ricadere in parte negli svantaggi del vecchio modello. In effetti molti degli esempi che si vedono anche su libri e articoli o in esempi di codice disponibili su Internet, tendono a usare poco i vantaggi di questo modello, restando legati al vecchio modo (non solo di Java) di concepire la gestione degli eventi. Di seguito i più diffusi errori di impostazione.

Spesso ad esempio i listener vengono implementati dallo stesso container in cui sono inseriti i componenti che generano l'evento. Questo si ricollega a un modo di concepire la gestione degli eventi piuttosto radicato nella mentalità di molti, che risale al C e al Visual Basic.

I componenti all'interno della finestra sono visti come se fossero degli "attributi" della finestra stessa e di cui è quindi la finestra a gestire il comportamento. Si tratta di una concezione palesemente contraria ai princìpi object oriented che favorisce la creazione di classi di dimensioni e complessità abnormi e la scrittura di codice involuto e difficilmente riusabile. Questa soluzione sarebbe da considerare utilizzabile solo in via eccezionale, in casi piuttosto semplici e in cui la riusabilità non sia importante.

Altre volte si tende a implementare i listener sempre e comunque come inner class (eventualmente anonime) del container o della finestra principale. È una impostazione che probabilmente si ricollega anch'essa al caso precedente: si tende cioè sempre a consi-

derare la gestione degli eventi come un “affare privato” di una finestra, solo per il fatto che, in effetti, “è lì” che accadono gli eventi. Questo però non significa affatto che sia “lì” che deve stare il codice che gestisce gli eventi.

Si è già dimostrata l’utilità di un listener implementato come classe esterna: la classe `MainWindowListener`, usata per gestire la chiusura della finestra principale, è stata utilizzata in quasi tutti gli esempi di questo capitolo. Invece, nella maggior parte degli esempi di codice che si possono comunemente incontrare nella pratica, si nota che listener anche uguali o molto simili tra loro vengono reimplementati ogni volta come classi interne. Anche qui è quasi superfluo osservare che si va contro i principi OO, rinunciando ai vantaggi che questo modello offre. In realtà la possibilità di implementare listener esterni riusabili *è uno dei principali vantaggi del delegation model* e dovrebbe essere sfruttato adeguatamente.

I ruoli nella gestione degli eventi

Per far funzionere il meccanismo degli eventi, viene attuata una stretta collaborazione fra tre tipi di oggetti: componenti, eventi e listener, che si appoggiano al “motore” della GUI, attraverso la JVM. Ai nostri fini non occorre distinguere tra JVM e sistema sottostante; si parlerà quindi di “sistema grafico” per riferirsi genericamente al “motore” esterno ai componenti, che fa funzionare tutto il sistema.

Il sistema grafico

Gli eventi sono generati normalmente dal sistema grafico e, in AWT, sono rappresentati da una lista (detta *coda degli eventi*) di oggetti `AWTEvent` che vengono inviati uno dopo l’altro ai componenti. In precedenza si è parlato di *componenti che originano un evento*; questa espressione si riferisce al fatto che, dal punto di vista dell’utente della GUI, l’evento ha origine da un’azione svolta sul componente, ad esempio un clic su un pulsante, e quindi si può dire che quel componente “ha generato” un evento. Da un punto di vista più vicino all’implementazione, tuttavia, è più corretto dire che il componente (l’oggetto Java) è il *target* dell’evento generato dal sistema grafico: è infatti il sistema grafico che esamina l’evento di input (ad esempio un clic del mouse), rileva che l’evento si riferisce a un determinato componente (quello tre i componenti visibili entro la cui area ricade il punto in cui è avvenuto il clic), quindi lo notifica al componente stesso.

I componenti

Di seguito, nei paragrafi successivi sono elencati i compiti svolti, nella gestione degli eventi, dal componente. Per ciascun compito sono indicati i relativi metodi della classe `Component`.

Generazione degli eventi

Si è visto che questo compito in genere è svolto internamente dal sistema grafico, spesso in seguito a input dell'utente. È però anche possibile generare un evento direttamente dal programma, utilizzando il metodo:

```
void dispatchEvent(AWTEvent e)
```

Questo metodo invia al componente l'evento dato in argomento, che sarà quindi elaborato come gli altri eventi.

Elaborazione degli eventi

```
void processEvent()
void processComponentEvent()
void processFocusEvent()
void processKeyEvent()
void processMouseEvent()
void processMouseMotionEvent()
```

Tutti questi metodi sono utilizzati dall'utente solo per la creazione di componenti custom. Sono i metodi che gestiscono gli eventi chiamando i metodi appropriati dei listeners registrati. I metodi si riferiscono agli eventi gestibili da *tutti* i componenti, quindi sono implementati nella classe base. Le sottoclassi hanno poi dei metodi `processXxxEvent` per ogni altro evento particolare che gestiscono.

```
void enableEvents(long eventMask)
void disableEvents(long eventMask)
```

Anche questi metodi sono protected e usati nella definizione di componenti custom. Servono per abilitare o disabilitare la gestione di un certo tipo di eventi. Un tipo di evento viene comunque abilitato quando viene aggiunto un listener. Con questi metodi è possibile abilitare o disabilitare una gestione interna degli eventi indipendentemente da quella dei listeners. L'argomento `inputMask` è una delle costanti definite nella classe `AWTEvent` che indicano un tipo di evento con un valore numerico (si veda più avanti).

Gestione dei listener

```
void addComponentListener(ComponentListener l)
```

```
void addFocusListener(focusListener l)

void addKeyListener(KeyListener l)

void addMouseListener(MouseListener l)

void addMouseMotionListener(MouseMotionListener l)
```

Tutti questi metodi servono per aggiungere un listener, relativo a un certo tipo di evento. Questi metodi si riferiscono agli eventi gestiti da tutti i componenti. Le sottoclassi hanno altri `metodi addXxxListener()` per altri tipi di eventi.

```
void removeComponentListener(ComponentListener l)

void removeFocusListener(focusListener l),

void removeKeyListener(KeyListener l)

void removeMouseListener(MouseListener l)

void removeMouseMotionListener(MouseMotionListener l)
```

Tutti questi metodi servono per rimuovere un listener, relativo a un certo tipo di evento. Questi metodi si riferiscono agli eventi gestiti da tutti i componenti. Le sottoclassi hanno altri metodi `removeXxxListener()` per altri tipi di eventi.

Gli oggetti `AWTEvent`

Gli oggetti evento, tutti appartenti a sottoclassi di `AWTEvent`, si occupano di mantenere le informazioni relative all'evento stesso. Ogni classe rappresenta in raltà un insieme di eventi generati dal sistema grafico affini tra loro. Ad esempio l'oggetto `KeyEvent` rappresenta gli eventi `KEY_PRESSED`, `KEY_RELEASED` e `KEY_TYPED`, generati rispettivamente quando un tasto viene premuto, rilasciato e quando avviene l'input di un carattere (risultante dall'azione di uno o più tasti). I singoli eventi sono rappresentati, nell'oggetto `event`, da costanti numeriche di identificazione, mentre la famiglia di eventi è rappresentata da una `event mask` ossia un valore numerico combinabile con l'operatore `or` usato ad esempio coi metodi `enableEvent()` e `disable Event()`.

In genere l'ID dell'evento (restituito dal metodo `getId()`) non è utilizzato dal client, poiché ad ogni singolo evento corrisponde uno specifico metodo del listener, quindi quando si scrive un gestore di un evento (ossia un metodo del listener) si sa già di quale specifico evento si tratta. I codici di identificazione sono usati in genere dai metodi di processing degli eventi dei componenti, che in base al codice chiamano i corrispondenti metodi del listener.

L'oggetto `AWTEvent` contiene anche alcuni metodi che invece sono spesso usati nei listener:

```
Object getSource()
```

Questo metodo è ereditato dalla classe base `EventObject` (gli eventi infatti non sono usati solo in AWT). Restituisce il "sorgente" (o, se si vuole il "target") dell'evento, ossia il componente a cui l'evento si riferisce. Poiché il valore di ritorno è un generico `Object`, è necessario fare un `cast` a `Component` o a una sua sottoclasse se si vogliono ottenere informazioni sul componente stesso. Questo metodo risulta utile in particolare nel caso in cui uno stesso listener sia collegato a più componenti. In questo caso questo metodo rende possibile l'identificazione di quale tra i componenti collegati sia il sorgente dell'evento.

```
void consume()
```

Questo metodo mette fine all'elaborazione dell'evento. Eventuali altri listener collegati all'evento non riceveranno la notifica. Un uso abbastanza comune di questo metodo è il filtraggio di caratteri di input: se si vuole che un carattere non venga elaborato, questo viene "consumato" dal gestore dell'evento `KEY_PRESSED`.

```
boolean isConsumed()
```

Indica se l'evento è stato "consumato".

Questi sono i metodi comuni a tutti gli eventi. Ogni sottoclasse di `AWTEvent` ha poi dei suoi metodi relativi a specifiche caratteristiche di quel tipo di evento. Ad esempio la classe `KeyEvent` ha, tra gli altri, un metodo `getKeyCode()` che restituisce il codice del tasto che è stato premuto, mentre la classe `MouseEvent` ha dei metodi che restituiscono le coordinate del punto in cui è avvenuto l'evento del mouse, e così via.

I listener

I listener sono gli oggetti che implementano i gestori degli eventi. Ogni listener, come si è già detto, rappresenta un insieme di eventi dello stesso tipo, e ad ogni specifico evento corrisponde un metodo del listener. I listener in AWT sono delle semplici interfacce, quindi possono essere implementate da qualunque classe. Tutti i listener. ad eccezione di quelli che hanno un unico metodo, hanno un corrispondente `adapter`, una classe che implementa tutti i metodi dell'interfaccia come metodi vuoti. Gli adapters sono utili se si vuole gestire solo un evento (o solo alcuni) di quelli gestiti dal listener: derivando una nuova classe dall'adapter si evita di dover implementare tutti i metodi dell'interfaccia anche se i corrispondenti eventi non vengono gestiti, poiché l'implementazione vuota di default è fornita dalla classe base.

Tabella 9.1 – *Listener e loro metodi*

Listener Interface	Adapter Class	Methods
`ActionListener`		`actionPerformed`
`Adjustmentlistener`		`adjustmentValueChanged`

`ComponentListener`	`ComponentAdapter`	`componentHidden`
		`componentMoved`
		`componentResized`
		`componentShown`

`ContainerListener`	`ContainerAdapter`	`componentAdded` `componentRemoved`
`FocusListener`	`FocusAdapter`	`focusGained` `focusLost`
`ItemListener`		`itemStateChanged`

`KeyListener`	`KeyAdapter`	`keyPressed`
		`keyReleased`
		`keyTyped`

`MouseListener`	`MouseAdapter`	`mouseClicked`
		`mouseEntered`
		`mouseExited`
		`mousePressed`
		`mouseReleased`

`MouseMotionListener`	`MouseMotionAdapter`	`mouseDragged` `mouseMoved`
`TextListener`		`textValueChanged`

`WindowListener`	`WindowAdapter`	`windowActivated`
		`windowClosed`
		`windowClosing`
		`windowDeactivated`
		`windowDeiconified`
		`windowIconified`
		`windowOpened`

Tabella 9.2 – *Componenti ed eventi generati*

Componente	Tipi di evento										
	action	adjustment	component	container	focus	item	key	mouse	mouse motion	text	window
`Button`	X		X		X		X	X	X		
`Canvas`			X		X		X	X	X		
`Checkbox`			X		X	X	X	X	X		
`CheckboxMenuItem`	*					X					
`Choice`			X		X	X	X	X	X		
`Component`			X		X		X	X	X		
`Container`			X	X	X		X	X	X		
`Dialog`			X	X	X		X	X	X		X
`Frame`			X	X	X		X	X	X		X
`Label`			X		X		X	X	X		
`List`	X		X		X	X	X	X	X		
`MenuItem`	X										
`Panel`			X	X	X		X	X	X		
`Scrollbar`		X	X		X		X	X	X		
`scrollPane`			X	X	X		X	X	X		
`TextArea`			X		X		X	X	X	X	
`TextComponent`			X		X		X	X	X	X	
`TextField`	X		X		X		X	X	X	X	
`Window`			X	X	X		X	X	X		X

* `CheckboxMenuItem` eredita il metodo `addActionListener()` da `MenuItem`, ma non genera action events.

Gestione degli eventi in AWT

In questa sezione sarà presentato un quadro panoramico della gestione degli eventi in AWT sotto forma di tabelle. Per maggiori dettagli consultare la documentazione dell'API.

I listener e i loro metodi

La tabella 9.1 mostra i vari listener e i metodi di ciascuno. Ogni metodo gestisce il corrispondente evento.

I componenti e gli eventi

La tabella 9.2 mostra invece quali eventi sono generati dai diversi componenti.

Interazioni tra gestori di eventi

Eventi e thread

Si è visto che gli eventi vengono messi in una coda ed eseguiti uno dopo l'altro. Ma cosa succede se due eventi vengono generati da thread differenti? Niente di particolare, poiché in AWT tutti gli eventi vengono gestiti in un unico thread chiamato event-dispatcher thread. In tal modo si evitano problemi di sincronizzazione dato che non c'è pericolo che la gestione di due eventi si sovrapponga. Possono sorgere problemi solo se si manipolano componenti al di fuori degli gestori di eventi (listeners).

Interazioni bidirezionali

Una interazione bidirezionale tra due componenti, si verifica in una situazione di questo tipo:

- viene generato un evento di tipo E1 su un componente C1;
- un gestore di E1 esegue del codice che a sua volta genera un evento di tipo E2 su un componente C2;
- un gestore di E2 esegue del codice che genera un evento di tipo E1 su C1.

A questo punto, se non c'è un sistema specifico per evitare una ricorsione, i passi 2 e 3 continuano a richiamarsi a vicenda all'infinito, bloccando il programma (o addirittura il sistema, per esaurimento di memoria).

Un caso comune in cui si rischia di incontrare questo problema è quello in cui si vogliono mantenere mutuamente aggiornati due componenti che mostrano due viste degli stessi

dati, entrambi modificabili dall'utente. Si è incontrato un caso del genere nell'esempio sulle scroll-bar, nel file `SliderPane.java`. In quell'esempio i due componenti collegati erano una `Scrollbar` e un `TextField`, che rappresentavano entrambi un valore numerico, mostrato nel primo con la posizione dello *scroll knob*, nel secondo direttamente come valore numerico. Entrambi i componenti potevano essere modificati dall'utente e il cambiamento dell'uno deve determinare un appropriato cambiamento dell'altro. In questo caso però non si genera alcuna ricorsione perché l'evento `AdjustmentEvent` della `Scrollbar` si verifica solo in caso di modifica da parte dell'utente e non con ill metodo `setValue()`, usato dal gestore dell'evento `TEXT_CHANGE`.

Il seguente esempio mostra un caso leggermente diverso in cui le due viste sono rappresentate da due `TextField` che mostrano la stessa misura lineare in due unità di misura differenti (centimetri e inches, *pollici*). Nel caso del `TextField`, il metodo `setText()` non si limita ad assegnare il testo, ma genera anche un evento `TEXT_VALUE_CHANGED`. Quindi, se non si prendono delle precauzioni, si verificherà la ricorsione descritta sopra.

```
import java.awt.*;
import java.awt.event.*;

public class UnitConverter extends Frame {
   TextField cmTextField = new TextField(20);
   TextField inchTextField = new TextField(20);
   ConvertionListener cmToInchListener
   = new ConvertionListener(inchTextField, new CmToInch());
   ConvertionListener inchToCmListener
   = new ConvertionListener(cmTextField, new InchToCm());

   public UnitConverter() {
      setBounds(100, 100, 230, 90);
      setTitle("Convertitore cm-inch");
      addWindowListener(new MainWindowListener());
      setLayout(new GridLayout(2, 1));
      add(createCmPanel());
      add(createInchPanel());
   }

   Panel createCmPanel() {
      Panel p = new Panel(new FlowLayout(FlowLayout.RIGHT));
      cmToInchListener.setConnectedListener(inchToCmListener);
      cmTextField.addTextListener(cmToInchListener);
      p.add(new Label("cm"));
      p.add(cmTextField);
      return p;
   }

   Panel createInchPanel() {
```

```
      Panel p = new Panel(new FlowLayout(FlowLayout.RIGHT));
      inchToCmListener.setConnectedListener(cmToInchListener);
      inchTextField.addTextListener(inchToCmListener);
      p.add(new Label("inch"));
      p.add(inchTextField);
      return p;
   }

   public static void main(String[] args) {
      new UnitConverter().show();
   }
}

class NumberFilter extends KeyAdapter {
   public void keyTyped(KeyEvent e) {
      char c = e.getKeyChar();
      if (!Character.isDigit(c)
            && !Character.isISOControl(c) && c != '.')
         e.consume();
   }
}

class ConvertionListener implements TextListener {
   TextField connectedTextField;
   ConvertionListener connectedListener;
   Converter converter;
   boolean enabled = true;

   ConvertionListener(TextField tf, Converter conv) {
      connectedTextField = tf;
      converter = conv;
   }

   public void textValueChanged(TextEvent e) {
      if (enabled) {
         TextField source = (TextField)e.getSource();
         String text = source.getText();
         String convertedText = null;
         try {
            converter.setValue(Double.parseDouble(text));
            convertedText = Double.toString(converter.getConvertedValue());
         }
         catch (NumberFormatException ex) {
         }
         // riga da commentare
         connectedListener.setEnabled(false);
         connectedTextField.setText(convertedText);
      }
      else
```

```
        enabled = true;
   }

   void setEnabled(boolean b) {
      enabled = b;
   }

   void setConnectedListener(ConvertionListener l) {
      connectedListener = l;
   }
}

abstract class Converter {
   double value;

   public void setValue(double value) {
      this.value = value;
   }

   abstract double getConvertedValue();
}

class CmToInch extends Converter {
   double getConvertedValue() {
      return value / 2.54;
   }
}
class InchToCm extends Converter {
   double getConvertedValue() {
      return value * 2.54;
   }
}
```

Per gestire questa situazione viene qui utilizzata una classe `ConversionListener` che mantiene due reference all'altro `TextField` e al suo listener (che deve essere anch'esso un `ConversionListener`) e un flag che indica lo stato abilitato/disabilitato. Quando viene gestito l'evento, il metodo `textValueChanged()`, prima di modificare l'altro `TextField`, disabilita il suo listener. Sarà poi lo stesso listener a reimpostare a `true` il suo flag `enabled` quando verrà eseguito il metodo con il flag impostato a `false`. In questo modo si evita che i due listener si richiamino l'un l'altro e il programma funziona correttamente. Se si prova a commentare la riga dove è riportato il commento, si noterà uno sfarfallio del testo e il cursore del testo rimarrà sempre ad inizio riga, mostrando che il metodo dei listener viene richiamato continuamente.

L'esempio mostra anche come effettuare il filtraggio dell'input dei `TextField` per scartare tutti i caratteri non numerici, per mezzo di un `KeyListener` che intercetta i

caratteri e "consuma" ossia scarta i caratteri non validi. I caratteri di controllo (ISO control) non vengono scartati, altrimenti si impedirebbe il funzionamento di tutti i tasti di spostamento del cursore, cancellazione ecc.

La grafica

Colori

In AWT il colore è rappresentato dalla classe `Color`, dotata di parecchi metodi che permettono di lavorare sui colori con grande flessibilità. Oltre ai soliti componenti RGB (red, green, blue) la classe supporta diversi metodi di rappresentazione dei colori (color space), come CMYK (cyan, magenta, yellow, black) o HSB (hue, saturation, brightness). Inoltre permette di specificare un valore di trasparenza (alpha).

Ecco un elenco dei metodi più comunemente usati.

Costruttori

```
Color(int red, int green, int blue)
```

Crea l'oggetto a partire dai valori dei componenti RGB indicati come interi nel range 0-255.

```
Color(int red, int green, int blue, int alpha)
```

Come sopra ma aggiunge il valore di trasparenza `alpha`; un valore di `alpha` uguale a `0` indica totale trasparenza, un valore di `255` totale opacità.

```
Color(float red, float green, float blue)
```

Qui i valori RGB sono specificati come `float` in un range da `0` a `1`, permettendo un maggior numero di sfumature.

```
Color(float red, float green, float blue, float  alpha)
```

Come sopra con il valore di trasparenza sempre nel range 0-1

```
Color(int rgb)
```

Interpreta il valore intero come valore RGB, in cui il primo byte rappresenta il valore red, il secondo green, il terzo blue.

Il seguente non è un costruttore, ma un factory method statico, che genera un colore a partire dai valori HSB:

```
static Color getHSBColor(int hue, int saturation, int brightness)
```

I valori vanno specificati nel range 0-255.

Componenti

```
int getRed()
int getGreen()
int getBlue()
int getAlpha()
```

Restituiscono le singole componenti del colore. Altri metodi restituiscono i componenti in altri spazi non RGB.

Variazione di luminosità

```
Color getBrighter()
Color getDarker()
```

Restituiscono una versione con luminosità maggiore o minore del colore dell'oggetto al quale i metodi appartengono. Il grado di luminosità aggiunto o tolto è arbitrario.

Conversione di colori

```
static int HSBtoRGB(float hue, float saturation, float brightness)
```

In questo caso i valori HSB sono specificati come float nel range 0-1 e il valore RGB risultante è nella forma compatta in un unico intero.

```
static float[] RGBtoHSB(int red, int green, int blue, float[] hsb)
```

I valori HSB, nel range 0-1, vengono inseriti nell'array dato come argomento, che è anche restituito come valore di ritorno. Se l'argomento è `null`, viene creato e restituito un nuovo array.

La classe `Color` ha una sottoclasse chiamata `SystemColor`, che viene usata per i colori definiti nell'ambiente grafico come default per i vari elementi grafici. Poiché in questo caso i valori RGB dipendono dalle impostazioni correnti, il colore non ha un valore fisso.

Oltre a poter specificare un colore per mezzo di una combinazione di valori dei suoi componenti, è possibile utilizzare una serie di oggetti statici delle classi `Color` e `SystemColor`.

La classe Color definisce 13 colori base:

```
black, blue, cyan, darkGray, gray, green, lightGray,
magenta, orange, pink, red, white, yellow
```

La classe `SystemColor` definisce invece 26 colori di sistema:

```
activeCaption, activeCaptionBorder, activeCaptionText,
inactiveCaption, inactiveCaptionBorder, inactiveCaptionText
control, controlText, controlShadow, controlDKShadow,
controlHighlight, controlLtHighlight
desktop
info, infoText
menu, menuText
scrollbar
text, textHighlight, textText, textHighlightText, textInactiveText
window, windowBorder, windowText
```

Da notare che, contrariamente al solito, i nomi di queste costanti sono in minuscolo. Per maggiori dettagli, consultare la documentazione dell'API.

I metodi della classe `Component` che utilizzano i colori sono:

```
void setBackground(Color c)
Color getBackground()
```

Assegnano e restituiscono il colore di sfondo.

```
void setForeground(Color c)
Color getForeground()
```

Assegnano e restituiscono il colore di default usato per disegnare testo e linee.

Font

La classe `Font` è stata notevolmente estesa nel JDK 1.2. Ci si si limita qui a descrivere le funzionalità comprese nel JDK 1.1.

Costruttore

```
Font(String name, int style, int size)
```

Crea un font con un determinato nome, stile e dimensione; questi valori restano fissi per tutta la durata dell'oggetto e non possono essere modificati in seguito.

Nome e famiglia

```
String getName()
string getFamily()
```

Restituiscono il nome del font (ad esempio Courier bold) e la famiglia di appartenenza (ad esempio Courier).

Per avere un elenco dei font presenti nel sistema si può usare, in Java 1.1, un metodo della classe `Toolkit` (una classe usata di solito internamente da AWT), come nel seguente esempio.

```
String[] fontNames = Toolkit.getDefaultToolkit().getFontList()
```

È piuttosto importante ricordare che in Java 2 questo metodo è deprecato. Al suo posto, con lo stesso fine, sarà necessario impiegare il metodo `GraphicsEnvironment.getAvailableFontFamilyNames()`.

Stile

```
int getStyle()
boolean isPlain()
boolean isBold()
boolean isItalic()
```

Il valore dello stile può essere una delle costanti numeriche `PLAIN`, `BOLD` e `ITALIC` oppure `BOLD + ITALIC`. I metodi booleani verificano lo stato delle singole proprietà di stile.

Dimensione

```
int getSize()
```

La dimensione è specificata in *punti tipografici*. Un punto corrisponde all'incirca a 1/72 di inch (approssimativamente 0.35 mm).

Un'altra classe utile per la manipolazione del testo è `FontMetrics`. Questa classe restituisce una serie di misure in pixel relative a un determinato font. Ecco un breve elenco di alcuni metodi di questa classe.

`FontMetrics(Font f)`

Costruisce un oggetto `FontMetrics` basato su uno specifico oggetto `Font`.

`getHeight()`

Restituisce l'altezza in pixel dei caratteri del font.

`getWidth(char c)`

Restituisce la larghezza in pixel di un determinato carattere del font.

`getWidth(String s)`

Restituisce la larghezza complessiva in pixel di una stringa.

La classe `Graphics`

La classe `Graphics` svolge fondamentalmente tre tipi di funzioni: svolge tutte le operazioni — saranno illustrate nei successivi paragrafi — di disegno di figure geometriche e di testo sull'area di un componente (o sulla stampante); rappresenta il *contesto grafico* di un componente, ossia mantiene una serie di parametri che vengono utilizzati per le operazioni di disegno; rappresenta il dispositivo grafico stesso (schermo, stampante, ecc.) a cui viene automaticamente collegato alla sua creazione. Quindi se si usano i metodi di un oggetto `Graphics`, le operazioni di disegno danno origine a una operazione di scrittura su tale dispositivo grafico.

Il contesto grafico

Le informazioni di contesto mantenute da un oggetto `Graphics` sono:

- L’oggetto `Component` al quale si riferisce, su cui vengono effettuate le operazioni grafiche.

- Un punto d’origine a cui si riferiscono le coordinate nelle operazioni di disegno, che può non coincidere con l’origine dell’area del componente.

- Le coordinate di un *clip rectangle*, ossia un rettangolo che delimita l’area di disegno (eventuali operazioni di disegno richieste al di fuori di quest’area non vengono eseguite).

- Il colore corrente, utilizzato per ogni operazione di disegno.

- Il font corrente, utilizzato per ogni operazione di visualizzazione di testo.

- La modalità di disegno (XOR o Paint) e il colore di alternanza usato per il disegno in modalità XOR.

La modalità Paint è quella normale: i disegni vengono effettuati con il colore corrente e si sovrappongono al contenuto corrente dell’area del componente. In modalità XOR i pixel dell’area di disegno vengono modificati secondo l’algoritmo `pixelColor = pixelColor ^ altColor ^ currentColor`, ossia i valori numerici che rappresentano i colori dell’area di disegno, il colore corrente e quello di alternanza, sono combinati con una operazione XOR bit a bit. Il risultato è che i pixel che sono del colore corrente vengono modificati nel colore di alternanza, mentre quelli del colore di alternanza diventano del colore corrente. Gli altri pixel vengono modificati dando origine a varie gradazioni di colore. Ma il fatto rilevante in tutto questo è che due operazioni di disegno in questa modalità si annullano a vicenda, ripristinando i colori originali, grazie a una proprietà dell’operatore XOR. Questo rende la modalità `XOR` molto utile quando si vogliono disegnare le cosiddette *ghost shapes* ossia dei disegni temporanei di figure geometriche che seguono i movimenti del mouse (usati in molti programmi grafici per disegnare figure o per selezionare aree).

Figure geometriche

La classe `Graphics` fornisce metodi per il disegno delle seguenti figure: linee; insiemi di linee congiunte (polyline); rettangoli; rettangoli con angoli arrotondati; rettangoli “tridimensionali”, ossia con i bordi che simulano un rilievo; poligoni; ellissi (oval); archi. Tutte le figure possono essere disegnate normalmente (solo il contorno) oppure riempite internamente con il colore corrente. Non esiste un metodo per disegnare un singolo punto, quindi per compiere questa operazione è necessario usare un altro metodo (ad esempio `drawLine()` specificando una lunghezza di 1 pixel).

Lo spessore delle linee è fisso, e uguale a 1 pixel. Non esistono stili delle linee (ed asempio tratteggiata, a punti, punto e linea ecc.) ma le linee vengono sempre disegnate come linee continue.

Come si può constatare, le capacità grafiche della classe `Graphics` sono piuttosto limitate. Per ottenere maggiori funzionalità e maggiore flessibilità bisogna ricorrere alla sottoclasse `Graphics2D`, aggiunta nel JDK 1.2, e concepita per applicazioni che fanno un uso intenso della grafica. Questa classe non verrà descritta in questa sede, trattandosi di una classe specializzata per applicazioni grafiche.

Testo

Con la classe `Graphics` è possibile disegnare del testo utilizzando il metodo `drawString()`, che prende come argomenti la stringa da visualizzare, e le coordinate di origine della stringa. Esistono anche altri metodi che prendono come argomenti un array di char o di byte. Nessun'altra operazione sul testo è supportata dalla classe `Graphics`. La classe `Graphics2D` include una serie di altri metodi che permettono di disegnare delle stringhe provviste di attributi (ad esempio con font o colori variabili per diversi caratteri).

Le immagini

Le immagini sono oggetti grafici che rappresentano, appunto, un'immagine per mezzo di un array di bits, interpretati in modo diverso a seconda del numero di colori e del formato specifico dell'immagine, ma che comunque rappresentano i colori di ciascun pixel. Il toolkit di default dovrebbe riconoscere almeno i formati GIF e JPEG.

Le immagini sono gestite in AWT per mezzo delle classi `Toolkit`, `Image`, `Graphics` e `Component` e dell'interfaccia `ImageObserver`. Diamo qui di seguito una breve descrizione del ruolo di ciascuna classe e di alcuni dei metodi coivolti:

`Toolkit`

La classe `Toolkit` crea e fornisce ai componenti le risorse di tipo immagine, con svariati procedimenti. I metodi più comunemente usati sono:

```
Image getImage(String fileName)
```

Crea e restituisce un'immagine a partire dal pathname completo di un file grafico in un formato riconosciuto.

```
Image getImage(URL imageURL)
```

Crea e restituisce un'immagine a partire da una URL.

Il metodo `getImage` restituisce un'unica istanza di `Image` per ogni specifica risorsa (due chiamate a `getImage()` con lo stesso argomento `fileName` restituiscono un reference alla stessa istanza di `Image`). Se invece si vogliono oggetti distinti si possono usare i corrispondenti metodi `createImage()`.

Image

Questa classe astratta rappresenta una generica immagine. Gli oggetti concreti sono creati direttamente o indirettamente dal `Toolkit`.

Tra i metodi della classe `Image` si citano

```
int getWidth(ImageObserver observer)
```

```
int getHeight(ImageObserver observer)
```

Restituiscono le dimensioni dell'immagine in pixel. Per il ruolo dell'argomento `observer` vedere più avanti la descrizione dell'interfaccia `ImageObserver`.

Graphics

La classe `Graphics` contiene diverse versioni del metodo `drawImage()` di cui si riportano le versioni con il minore e con il maggiore numero di argomenti:

```
void drawImage(Image image, int x, int y, ImageObserver observer)
```

Questi argomenti devono essere sempre specificati: l'immagine da visualizzare, le coordinate dell'origine (angolo superiore sinistro), e un oggetto che implementa l'interfaccia `ImageObserver`.

```
void drawImage(Image image, int x, int y, int width,
               int height, Color background, ImageObserver observer)
```

Gli argomenti opzionali sono: le dimensioni dell'immagine (se specificate, l'immagine viene ingrandita o resa più piccola per adattarsi alle dimensioni specificate) e il colore di background, con il quale vengono resi gli eventuali pixel trasparenti.

ImageObserver

L'interfaccia `ImageObserver`, appartenente al package `java.awt.image`, viene uti-

lizzata per la gestione del caricamento dell'immagine, che avviene in maniera asincrona. Ciò significa che una chiamata a un metodo `getImage()` termina subito restituendo un oggetto `Image` che non è ancora inizializzato, per cui le sue proprietà non sono ancora disponibili. Il caricamento dell'immagine avviene solo quando il client ne richiede il disegno oppure una proprietà. Anche questi metodi (ad esempio `Graphics.drawImage()` o `Image.getWidth()`) terminano immediatamente indipendentemente dallo stato effettivo dell'oggetto `Image`. Il metodo `getWidth()` restituisce `-1` se la proprietà non è ancora disponibile e in questo caso dà inizio al caricamento dell'immagine su un thread separato. Lo stesso farà il metodo `drawImage()` se l'immagine non è stata precedetemente caricata, per poi procedere, sempre nel thread separato, alla visualizzazione.

Sia nel caso della proprietà che in quello del disegno, il client potrà avere comunque necessità di accertare quando il caricamento ha avuto luogo (per ottenere il valore effettivo della proprietà o per compiere operazioni che presuppongono l'avvenuto caricamento dell'immagine). A questo provvede il metodo

```
boolean imageUpdate(Image image, int infoFlags,
                    int x, int y, int width, int height)
```

Il metodo viene chiamato dal codice che esegue il caricamento dell'immagine ogni volta che nuove proprietà risultano disponibili. `imageUpdate()` restituisce un boolean che deve essere `false` se tutte le proprietà necessarie sono state ottenute, `true` se invece sono richieste ulteriori informazioni. L'argomento `infoFlags` è un insieme di flags combinati con l'operatore `OR`, che indicano quali informazioni sono disponibili. I singoli flag sono rappresentati da una serie di costanti definite nell'interfaccia stessa, che rappresentano le varie informazioni su un'immagine. Se ne descrivono brevemente alcune. `WIDTH`: la larghezza dell'immagine è disponibile; `HEIGHT`: l'altezza dell'immagine è disponibile; `ALLBITS`: tutti i dati che descrivono i pixel dell'immagine sono disponibili; `ERROR`: si è verificato un errore e il caricamento non può essere ultimato. Gli argomenti `width` e `height` indicano le rispettive proprietà, mentre `x` e `y` vengono usati quando i bit dell'immagine vengono resi disponibili (ed eventualmente disegnati) un po' alla volta.

`Component`

La classe `Component` implementa l'intefaccia `ImageObserver`, provvedendo alla visualizzazione dell'immagine man mano che si rende disponibile, qualora questa sia richiesta dal metodo `paint()`. Vi sono poi alcuni metodi che permettono di caricare preventivamente l'immagine o avere informazioni sul suo stato.

```
int checkImage(Image image, ImageObserver observer)
```

Restituisce i flags indicanti l'attuale stato dell'immagine, ma senza avviarne il caricamento.

```
void prepareImage(Image image, ImageObserver observer)

void prepareImage(Image image, int width, int height,
                  ImageObserver observer)
```

Avviano il caricamento dell'immagine (eventualmente ridimensionata secondo le dimensioni date), senza visualizzarla.

Infine ci sono alcuni metodi per la creazione di immagini.

```
Image createImage(int width, int height)
```

Crea un'immagine *off screen*, ossia immagini create in memoria per una successiva visualizzazione delle dimensioni specificate.

```
Image createImage(ImageProducer producer)
```

Crea un'immagine per mezzo di un oggetto `ImageProducer`. `ImageProducer` è un'interfaccia implementata dalle classi che provvedono alla creazione di immagini. Per maggiori dettagli consultare la documentazione API.

Il seguente esempio mostra la visualizzazione di un'immagine per mezzo di un componente custom specializzato, e le funzioni di ridimensionamento.

```
import java.awt.*;
import java.awt.image.*;
import java.math.*;

public class ImageComponent extends Label {
   String text;
   Image image;
   int originalImageWidth;
   int originalImageHeight;
   int scaleX = 100;
   int scaleY = 100;

   public ImageComponent(String fileName, String text) {
      this.text = text;
      setImage(fileName);
   }

   public void setImage(String fileName) {
      if (fileName == null) {
         setText(text);
         image = null;
```

```
      }
      else {
         setText(null);
         image = Toolkit.getDefaultToolkit().getImage(fileName);
         originalImageWidth = image.getWidth(this);
         originalImageHeight = image.getHeight(this);
      }
   }

   void adjustSize() {
      int textHeight = 0;
      int textWidth = 0;
      try {
         if (text != null) {
            FontMetrics fm = getFontMetrics(getFont());
            textHeight = fm.getHeight();
            textWidth = fm.stringWidth(text);
         }
         int width = Math.max(originalImageWidth * scaleX / 100, textWidth + 8);
         int height = originalImageHeight * scaleY / 100 + textHeight;
         setSize(width, height);
      }
      catch (NullPointerException e) {
         // se getFontMetrics() restituisce null
         // viene generata un eccezione che viene ignorata
      }
   }

   public boolean imageUpdate
   (Image image, int infoFlags, int x, int y, int width, int height) {
      if ((infoFlags & (ImageObserver.WIDTH | ImageObserver.HEIGHT)) != 0) {
         originalImageWidth = width;
         originalImageHeight = height;
         adjustSize();
         getParent().validate();
      }
      return super.imageUpdate(image, infoFlags, x, y, width, height);
   }

   public void setScale(int scaleX, int scaleY) {
      if (scaleX > 0)
         this.scaleX = scaleX;
      if (scaleY > 0)
         this.scaleY = scaleY;
      adjustSize();
      getParent().validate();
   }

   public Dimension getPreferredSize() {
```

```
        return getSize();
    }

    public void paint(Graphics g) {
        if (image == null)
            return;
        int imageWidth = originalImageWidth * scaleX / 100;
        int imageHeight = originalImageHeight * scaleY / 100;
        int textWidth = 0;
        int textHeight = 0;
        FontMetrics fm = getFontMetrics(getFont());
        if (text != null) {
            textHeight = fm.getHeight();
            textWidth = fm.stringWidth(text);
        }
        int x = (getSize().width - imageWidth) / 2;
        int y = (getSize().height - imageHeight - textHeight) / 2;
        if (text != null) {
            g.drawString(text, (getSize().width - textWidth) / 2, y
                          + imageHeight + fm.getAscent());
        }
        g.drawImage(image, x, y, imageWidth, imageHeight, this);
    }
}
```

Quello che fa il componente è semplicemente visualizzare un'immagine caricata da file e/o una stringa di testo. Se sono presenti entrambi, immagine e testo, quest'ultimo viene visualizzato sotto l'immagine. Se l'immagine non è presente, allora la visualizzazione viene demandata alla classe base. L'immagine, se presente, viene posizionata al centro dell'area del componente.

Il costruttore chiama il metodo `setImage()` che crea l'immagine o, se il nome del file passato come argomento è `null`, assegna il testo alla proprietà `text` della classe base (in tal modo la visualizzazione verrà gestita dal peer, indipendentemente dal metodo `paint()` ridefinito). Come si è detto, con il metodo `getImage()` l'immagine non viene caricata, ma viene solo creato l'oggetto `Image`. Subito dopo la creazione, vengono chiamati i metodi `getWidth()` e `getHeight()` dell'oggetto `image` che, a meno che l'immagine non sia già stata precedentemente creata (con una chiamata a `getImage()` per lo stesso file) restituiranno `-1`. Per garantire che il valore di `width` e `height` siano corretti al momento della visualizzazione, viene ridefinito il metodo `ImageUpdate()` che assegna i valori quando questi sono disponibili (dopodiché assegna al componente le dimensioni appropriate chiamando il metodo `adjustSize()` e forza il ricalcolo del layout del container con `getParent().validate()`). Il metodo chiama poi l'implementazione di `ImageUpdate` della classe base e restituisce il valore di ritorno di quest'ultimo. Questo perché si presuppone che il metodo della classe base non possa restituire `false` (met-

tendo fine alle chiamate callback di `ImageUpdate`) finché tutte le informazioni — comprese quindi `width` e `height` — non siano disponibili. Infatti senza di queste il metodo non potrebbe terminare il caricamento dell'immagine.

Il metodo `adjustSize()` assegna al componente le dimensioni sufficienti a visualizzare l'immagine (alla scala assegnata) e il testo.

Il metodo `setScale()` assegna i valori della scala orizzontale e verticale e aggiorna la visualizzazione. I valori vengono modificati solo se il valore passato è maggiore di `0` (questo consente di impostare uno solo dei due valori passando un valore `0` o negativo).

Il metodo `getPreferredSize()` è ridefinito per fornire ai `LayoutManager` un valore corretto.

Il metodo `paint()` infine, disegna l'immagine, tenendo conto dei valori di scala. Se le dimensioni del componente sono maggiori di quelle dell'immagine (nel caso che la dimensione venga modificata dal `LayoutManager`), questa viene posizionata al centro dell'area del componente. Il testo viene comunque visualizzato sotto la figura.

La classe seguente è un `Frame` di test che visualizza un componente `ImageComponent` e permette di modificarne la scala orizzontale e verticale.

```
import java.awt.*;
import java.awt.event.*;

public class ImageDemo extends Frame {
   public static final String WIDTH = "Larghezza";
   public static final String HEIGHT = "Altezza";
   ImageComponent image;
   int scaleX = 100;
   int scaleY = 100;

   ItemListener choiceListener = new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
         Choice choice = (Choice)e.getSource();
         String name = choice.getName();
         int value = Integer.parseInt(choice.getSelectedItem());
         if (name.equals(WIDTH))
            image.setScale(value, 0);
         else
            image.setScale(0, value);
      }
   };

   public ImageDemo() {
      setTitle("Image Demo");
      setBounds(100, 100, 320, 300);
      addWindowListener(new MainWindowListener());
      image = new ImageComponent("globe.gif", "Globe.gif");
      image.setBackground(new Color(0, 0, 64));
```

```
      image.setForeground(Color.white);
      Panel upperPanel = new Panel();
      upperPanel.setLayout(new GridLayout(1, 2));
      upperPanel.add(getScalePanel(WIDTH));
      upperPanel.add(getScalePanel(HEIGHT));
      add(upperPanel, BorderLayout.NORTH);
      add(image, BorderLayout.CENTER);
   }

   Panel getScalePanel(String name) {
      Panel p = new Panel();
      p.setLayout(new FlowLayout(FlowLayout.CENTER, 2, 4));
      p.add(new Label(name));
      Choice choice = new Choice();
      choice.setName(name);
      choice.addItem("50");
      choice.addItem("75");
      choice.addItem("100");
      choice.addItem("125");
      choice.addItem("150");
      choice.addItem("175");
      choice.addItem("200");
      choice.select("100");
      choice.addItemListener(choiceListener);
      p.add(choice);
      p.add(new Label("%"));
      return p;
   }

   public static void main(String[] args) {
      new ImageDemo().show();
   }
}
```

Nella parte superiore del `Frame` viene inserito un pannello che contiene altri due pannelli con due componenti `Choice`, uno per la scala orizzontale, uno per la scala verticale. I `Choice` permettono di selezionare valori compresi tra il 50% e il 200%. Il listener dei `Choice` (lo stesso per entrambi) chiama il metodo `setScale()` dell'`ImageComponent` passando gli argomenti a seconda del `Choice` sorgente dell'evento (riconosciuto attraverso il nome).

Il supporto per la stampa

AWT fornisce un semplice supporto per la stampa, che si appoggia sulle funzionalità del sistema grafico sottostante. Le classi e interfacce coinvolte nella gestione della stampa

sono la classe `Toolkit`, che dà inizio a una operazione di stampa e dà accesso a un oggetto `PrintJob`; la classe `PrintJob`, che mette a disposizione le risorse del sistema grafico e dirige le operazioni di stampa; la classe `Component`, che contiene alcuni metodi specifici per la stampa; l'interfaccia `PrintGraphics`, usata per gli oggetti `Graphics` che rappresentano il contesto grafico di stampa; la classe `Graphics`, della quale il `PrintJob` fornisce una sottoclasse che implementa l'interfaccia `PrintGraphics`.

`PrintJob`

Un'istanza della classe PrintJob si ottiene per mezzo del metodo

```
PrintJob Toolkit.getPrintJob(Frame dialogParent,
                              String title, Properties p)
```

Questo metodo dà inizio a una sessione di stampa nel sistema grafico, che in genere causa l'apertura di un *print dialog* che permette all'utente di selezionare le opzioni di stampa prima di avviarla. Quando l'utente avvia il processo di stampa dal dialog box, il metodo `getPrintJob()` termina restituendo un oggetto `PrintJob` che fornisce all'applicazione gli strumenti per il controllo e l'invio dei dati alla stampante. Se l'utente cancella la richiesta di stampa, oppure si verifica qualche errore, il metodo restituisce `null`.

Una volta ottenuto l'oggetto `PrintJob`, l'applicazione può cominciare a disegnare utilizzando l'oggetto `Graphics` che ottiene per mezzo del metodo

```
Graphics PrintJob.getGraphics()
```

L'oggetto rappresenta una singola pagina di stampa. I dati scritti attrverso i metodi dell'oggetto `Graphics` restano in memoria centrale finché non viene chiamato il metodo

```
void Graphics.dispose()
```

Questo metodo serve in generale per liberare le risorse del sistema grafico allocate dall'oggetto `Graphics`. Se l'oggetto è collegato a un dispositivo di stampa, causa anche un flush del buffer di stampa e i dati vengono solo a questo punto inviati effettivamente alla stampante. Una volta chiamato il metodo dispose, l'oggetto `Graphics` non è più utilizzabile, quindi sarà necessario richiedere un nuovo oggetto `Graphics` al `PrintJob` per ogni pagina.

Quando le operazioni di stampa sono terminate, l'applicazione *deve* rilasciare la risorsa con il metodo

```
void PrintJob.end()
```

In caso contrario potrà risultare impossibile utilizzare la stampante, che rimarrà soggetta al lock creato dall'applicazione.

`PrintJob` contiene anche i due metodi:

```
Dimension getPageDimension()
```

che restituisce le dimensioni della pagina, e

```
int getPageResolution()
```

che restituisce la risoluzione della stampante in pixel per inch. Su questi metodi si tornerà più avanti.

`Graphics` e `PrintGraphics`

Si sarà notato che il metodo `getGraphics()` di `PrintJob` restituisce un oggetto `Graphics` e non un `PrintGraphics`. In effetti `PrintGraphics` è semplicemente un'interfaccia dotata di un solo metodo `PrintJob getPrintJob()`, che permette al metodo che effettua le operazioni di stampa di ottenere un reference al `PrintJob`. Ma l'uso più comune di quest'interfaccia è stabilire se l'oggetto `Graphics` rappresenta una stampante o lo schermo, come nel seguente esempio

```
void draw(Graphics g) {
   boolean isPrinter = false;
   if (g instanceof PrintGraphics)
      isPrinter = true;

}
```

Il codice successivo potrà così effettuare operazioni diverse a seconda che il dispositivo grafico sia lo schermo o la stampante. Se poi al metodo occorre anche l'oggetto `PrintJob` (ad esempio per ottenere le dimensioni della pagina), può utilizzare un codice come questo

```
void draw(Graphics g) {
   PrintJob pj = null;
   if (g instanceof PrintGraphics)
      pj = ((PrintGraphics)g).getPrintJob();

}
```

Qui la variabile `pj` servirà sia a determinare il tipo di dispositivo (schermo se il valore è `null`, stampante in caso contrario), sia a mantenere un reference all'oggetto `PrintJob`.

Stampa di componenti

La classe `Component` implementa due metodi per le operazioni di stampa:

```
void print(Graphics g)
void printAll(Graphics g)
```

Il primo metodo stampa solo il componente, il secondo stampa il componente e, se questo è un container, stampa ricorsivamente tutti i componenti contenuti al suo interno.

L'implementazione di default del metodo `print()` chiama semplicemente il metodo `paint()`. Ridefinendo il metodo `print()` si può inserire del codice specifico per la stampa prima di chiamare `paint()`, o addirittura usare del codice totalmente differente da quello usato per disegnare sullo schermo. La scelta dipenderà, in generale, dall'entità delle differenze tra i due tipi di visualizzazione.

Per agevolare la stampa nel caso si usi lo stesso metodo di disegno (eventualmente con alcune variazioni a seconda del dispositivo di output) sia per il video che per la stampa, la risoluzione della pagina di stampa, restituita dal metodo

```
Dimension PrintJob.getPageDimension()
```

fornisce una dimensione virtuale che dovrebbe essere adattata alla risoluzione dello schermo attualmente in uso. In pratica la risoluzione data sembra tener ben poco conto dei parametri reali; ad esempio in un sistema Windows funzionante a risoluzione 1024×768 risulta una risoluzione di 595×842 che sembrerebbe piuttosto una impostazione fissa vicina ai valori minimi supportati da qualunque sistema grafico, in modo che possa funzionare con qualunque risoluzione. Stando così le cose l'utilità di questa conversione risulta piuttosto improbabile, visto che solo occasionalmente potrà (approssimativamente) corrispondere alla effettiva risoluzione del video, e quindi permettere una stampa senza necessità di riscalare le dimensioni dei componenti e dei loro contenuti. In ogni caso bisogna tener presente che la stampa viene sempre fatta con allineamento all'angolo superiore sinistro del foglio, quindi se si vuole centrare la stampa, o impostare un margine, si dovrà spostare l'origine dell'area di disegno con il metodo

```
void Graphics.translate(int x, int y)
```

Nella classe `Graphics` non esiste alcun metodo per ridefinire le dimensioni virtuali della pagina di stampa, che permetterebbe una flessibilità d'uso di gran lunga maggiore. Per ottenere questa e altre funzionalità avanzate bisogna ricorrere alla classe `Graphics2D`, disponibile a partire dal JDK 1.2, la cui trattazione non è prevista in questa sede.

Si deve inoltre tener presente che con i componenti che in teoria potrebbero prestarsi a una stampa diretta dei loro contenuti, come una `TextArea` o una `List`, difficilmente risulta utilizzabile la funzionalità di stampa diretta dei contenuti, utilizzando il metodo `print()` di default. Infatti se, come nella maggior parte dei casi, le dimensioni dell'area di disegno superano quelle dell'area visibile, la parte non visibile non sarà stampata. Dal momento che questi componenti non danno accesso ai sottocomponenti che effettivamente fanno da contenitori del testo (si tratta di oggetti gestiti direttamete dal sistema e non esposti all'utente) non risulta possibile stampare l'intero contenuto con il metodo di default. Questo invece è possibile se si inserisce un componente le cui dimensioni sono sufficienti a contenere tutto il testo e tale componente si inserisce in uno `ScrollPane`. Se in questo caso vienre mandato in stampa, il componente (e non lo `ScrollPane`) sarà stampato per intero anche se solo una sua parte risulta visibile sullo schermo.

Da quanto detto si può dedurre che, in pratica, raramente capita di stampare un componente usato per la visualizzazione su video, salvo il caso di componenti espressamente concepiti per la elaborazione di testi e/o immagini (ad esempio un ipotetico componente custom `TextEditor` usato per inserire testo formattato in vari font e stili) e dotato di specifico supporto per la stampa. In genere per la stampa in ambienti grafici vengono creati dei componenti ad hoc (chiamati generalmente *reporting components*) che permettono la composizione grafica delle pagine di stampa e riproducono più o meno esattamente l'aspetto del documento stampato.

Stampa di testo

Non disponendo di un supporto evoluto per la stampa, come quello fornito da componenti ad hoc, l'unica cosa da fare è gestire la stampa a basso livello. Se si deve stampare semplicemente del testo, questo va scritto direttamente sull'oggetto `Graphics` del `PrintJob`, dimensionandolo e posizionandolo opportunamente. L'esempio seguente mostra la stampa di un componente `FileTextArea`, sottoclasse di `TextArea`, che carica i contenuti di un file specificato e lo stampa, ridefinendo il metodo `print()`.

```
import java.awt.*;
import java.io.*;
import java.util.*;

public class FileTextArea extends TextArea {
   String title;

   public FileTextArea(String fileName, String title) {
      this.title = title;

      try {
         BufferedReader reader = new BufferedReader(new FileReader(fileName));
```

```
            String line = null;
            while ((line = reader.readLine()) != null)
               append(line + '\n');
         }
         catch (IOException e) {
            e.printStackTrace();
         }
   }

   public void print(Graphics g) {
      g.setFont(new Font("Arial", Font.BOLD, 24));
      FontMetrics fontMetrics = g.getFontMetrics();
      int lineHeight = fontMetrics.getHeight();
      g.translate(100, 100);
      g.drawString(title, 0, 0);

      int y = 2 * lineHeight;
      g.setFont(new Font("Arial", Font.PLAIN, 12));
      fontMetrics = g.getFontMetrics();
      lineHeight = fontMetrics.getHeight();

      StringTokenizer tokens = new StringTokenizer(getText(), "\r\n");
      int i = 0;
      while (tokens.hasMoreTokens()) {
         String s = tokens.nextToken();
         g.drawString(s, 0, y);
         y += lineHeight;
      }
   }
}
```

Il costruttore della `FileTextArea` prende due argomenti: il nome del file e il titolo da stampare, e inserisce al suo interno il testo del file.

Per semplicità, la stampa avviene con parametri fissi: margine superiore e sinistro di 100 punti, font Arial bold di dimensione 24 per il titolo e normale di dimensione 12 per il testo. I margini vengono impostati con il metodo `Graphics.translate()` e per stampare una riga si usa `Graphics.drawString()`, specificando le coordinate.

Poiché `drawString()` non gestisce correttamente i caratteri di andata a capo (o almeno non è garantito che lo faccia), le righe vengono separate in token con uno `StringTokenizer` e stampate una alla volta. Per determinare la coordinata `y` di ogni riga si usa l'altezza della riga, ottenuta con `FontMetrics.getHeight()`.

```
import java.awt.*;
import java.awt.event.*;

public class PrintDemo extends Frame {
```

```
    public PrintDemo() {
        setTitle("Print Demo");
        setBounds(100, 100, 300, 200);
        addWindowListener(new MainWindowListener());
        final FileTextArea textArea
        = new FileTextArea("LaVispaTeresa.txt", "La vispa Teresa");
        add(textArea, BorderLayout.CENTER);
        Button printButton = new Button("Stampa");
        printButton.addActionListener (
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    print(textArea);
                }
            }
        );
        add(printButton, BorderLayout.SOUTH);
    }

    public void print(Component component) {
        PrintJob printJob
        = Toolkit.getDefaultToolkit().getPrintJob(this, "Print Demo", null);
        if (printJob == null)
            return;
        try {
            Graphics g = printJob.getGraphics();
            component.print(g);
            g.dispose();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            printJob.end();
        }
    }

    public static void main(String[] args) {
        new PrintDemo().show();
    }
}
```

Il `Frame PrintDemo` contiene un componente `FileTextArea` e un pulsante per far partire la stampa. Il metodo `print` crea un `PrintJob` e chiama il metodo `print` della `FileTextArea` passandogli l'oggetto `Graphics` del `PrintJob`. Notare che la chiamata `printJob.end()` è inserita in un blocco `finally` per fare in modo che venga eseguita comunque, anche in caso di eccezioni, per evitare che la risorsa possa rimanere occupata in caso di malfunzionamenti.

Da prove effettuate in piattaforma Windows, solo il JDK 1.1 sembra gestire correttamente la stampa del testo. Utilizzando il JDK 1.2 viene generato un documento di stampa di dimensioni abnormi (circa 30 megabyte per un testo di meno di 400 caratteri) e la stampa non ha luogo o viene stampata solo l'ultima stringa scritta. Col JDK 1.2.2 la stampa viene effettuata correttamente, ma il documento di stampa ha ancora dimensioni eccessive, anche se sensibilmente minori rispetto al caso precedente (circa 2 megabyte). Solo con il JDK 1.1 la dimensione del documento di stampa assume proporzioni normali (meno di 5 kilobyte).

Componenti custom

In AWT è possibile definire nuovi componenti custom fondamentalmente in due modi: si possono definire sottoclassi di normali componenti come `Button`, `Label`, `Panel` ecc. ereditando il comportamento e l'aspetto grafico del componente base; si possono definire nuovi componenti partendo da zero come sottoclassi del componente `Canvas`, che è un componente base privo di funzionalità particolari.

Se si sceglie la seconda strada è praticamente d'obbligo ridefinire almento il metodo `paint()` che disegna il componente. Se si sceglie invece la prima, il metodo paint può essere o non essere ridefinito.

Se non viene ridefinito, l'aspetto grafico sarà quello del componente base e la personalizzazione riguarderà altri aspetti. Un esempio di questo tipo è il componente `FileTextArea` di un precedente esempio, che viene disegnato e si comporta come una normale `TextArea`, ma aggiunge il caricamento automatico di un testo da file e il supporto per la stampa del testo.

Se il metodo `paint()` viene ridefinito per un componente derivato da una classe diversa da `Canvas`, il disegno effettuato dal peer, cioè dal componente nativo del sistema grafico, viene comunque eseguito, anche se il metodo `paint()` della classe base non viene richiamato dal metodo ridefinito. Questo accade perché il peer, una volta creato, viene per certi aspetti gestito direttamente dal sistema grafico, senza che il codice Java possa modificarne il comportamento. A questo disegno si aggiungerà quello effettuato dal metodo `paint()` ridefinito.

La classe `ImageComponent`, definita nell'esempio precedente, è un componente custom derivato da `Label`. In questo caso la derivazione da questo componente si giustifica solo con il fatto che in tal modo un componente `ImageComponent` può essere utilizzato anche in tutte le situazioni in cui è richiesta una label (come label grafica). Qui il metodo `paint` del componente base viene semplicemente neutralizzato, mantenendo il valore della proprietà `text` a `null`. Il testo viene invece memorizzato su una nuova variabile di

classe in modo che la sua visualizzazione sia gestita esclusivamente dal metodo `paint()` ridefinito.

Nel caso specifico c'è anche un altro motivo per la scelta di `Label` come classe base: da prove effettuate in ambiente Windows, si è notato che il componente `ImageComponent`, se derivato da `Canvas`, nella visualizzazione di immagini GIF animate dà uno sgradevole sfarfallio, che invece scompare se il componente viene derivato da `Label`. Questo comportamento dipende dall'implementazione e dal sistema sottostante, ma conferma la difficoltà di previsione del comportamento di alcune caratteristiche dei componenti (custom e non) dovute all'implementazione nativa dei peer. Questo suggerisce che a volte, per ottenere i migliori risultati nell'implementare componenti custom AWT, può essere opportuno sperimentare diverse alternative.

I componenti Swing

DI ANDREA GINI

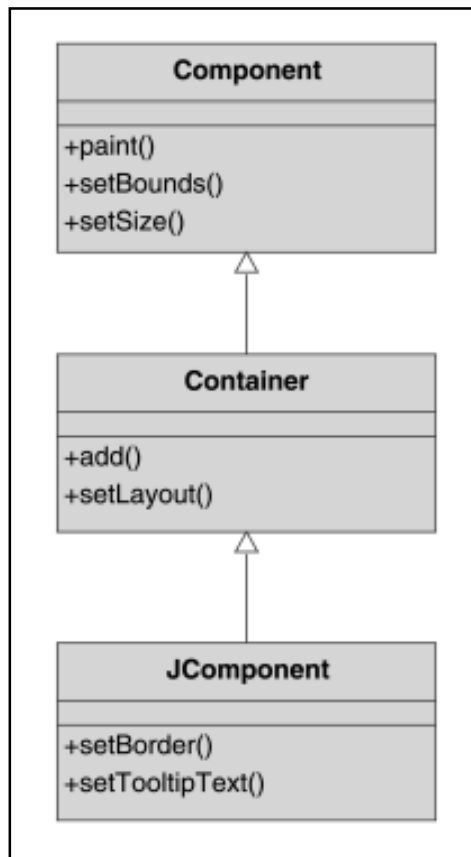
Introduzione: differenze tra Swing e AWT

Come è ormai noto, la sfida più grande affrontata dai progettisti di Java nello studio della JVM fu quello di realizzare un package grafico capace di funzionare con un buon livello di prestazioni su piattaforme molto differenti tra loro. La soluzione proposta nel 1996 fu AWT, un package che mappava i componenti grafici del sistema ospite con una serie di classi scritte in gran parte in codice nativo. Questa scelta limitava pesantemente la scelta dei componenti da includere nella API, poiché costringeva a prendere in considerazione solamente quel limitato insieme di controlli grafici che costituivano il "minimo comun denominatore" tra tutti i sistemi a finestre esistenti. Inoltre comportava grossi problemi di visualizzazione, poiché le interfacce grafiche realizzate originariamente su una JVM apparivano spesso con grossi difetti se eseguite su una JVM differente.

L'insieme di componenti Swing, al contrario, è stato realizzato totalmente in codice Java, senza ricorrere alle primitive di sistema. In questo modo sono stati risolti alla radice i problemi di compatibilità, visto che la stessa identica libreria viene utilizzata, senza alcuna modifica, su qualunque JVM. Dal momento che Sun Microsystem ha reso pubblico il codice delle API Java, è possibile addirittura studiare i sorgenti di un particolare componente, ad esempio di un pulsante, e scoprire attraverso quali metodi esso viene disegnato sullo schermo. Liberi dal vincolo del "minimo comun denominatore", i progettisti di Swing hanno scelto di percorrere la via opposta, creando un package ricco di componenti e funzionalità spesso non presenti nella piattaforma ospite.

Per risolvere definitivamente il problema della differenza di visualizzazione, viene offerta al programmatore la possibilità di scegliere il *Look & Feel* con cui visualizzare le proprie GUI, al contrario di quanto avveniva su AWT che era vincolato alla rappresentazione dei componenti della macchina ospite.

Figura 10.1 – *Pluggable Look & Feel*

Figura 10.2 – `JComponent` *eredita il proprio comportamento sia da* `Component` *che da* `Container`. *Di nuovo porta la capacità di aggiungere bordi e ToolTip*

Questa caratteristica, denominata Pluggable Look & Feel, verrà trattata in maniera approfondita al termine di questo capitolo. La presenza di queste features non va a incidere sulla caratteristica più richiesta da un programmatore: la facilità d'uso. Chi avesse già maturato una buona conoscenza di AWT, si troverà a proprio agio con Swing, data la sostanziale compatibilità tra l'interfaccia di programmazione dei rispettivi componenti. Chi invece fosse alla ricerca di nuove modalità operative, troverà soddisfazione nell'esplorare le caratteristiche più avanzate di questa straordinaria API grafica.

Il padre di tutti i componenti Swing: `JComponent`

I componenti Swing sono più di 70, e sono quasi tutti sottoclassi di `JComponent`, una classe astratta che ha `Component` e `Container` come antenati. Al pari di `Component`, `JComponent` è un oggetto grafico che ha una posizione e una dimensione sullo schermo, al cui interno è possibile disegnare, scrivere o ricevere eventi dal mouse e dalla tastiera. Come `Container` invece, offre la possibilità di disporre altri componenti al suo interno, nonché il supporto ai Layout Manager, oggetti che semplificano il lavoro di impaginazione delle interfacce grafiche. La classe `JComponent` fornisce in esclusiva alcune funzionalità, che sono presenti in ogni sua sottoclasse. Tra queste si possono segnalare:

ToolTips

Il metodo `setToolTipText(String)` permette di aggiungere un Tool Tip, un messaggio testuale che compare sul componente dopo che l'utente vi ha lasciato fermo il mouse per qualche istante.

Bordi

Ogni componente Swing può essere dotato di un bordo grazie al metodo `setBorder(Border)`. Il package `javax.Swing.Border` fornisce strumenti per creare diversi tipi di bordo, da semplici contorni a sofisticate decorazioni.

Pluggable Look and Feel

È la possibilità offerta da ogni `JComponent` di essere visualizzato in maniera differente a seconda del Look & Feel impostato dall'utente.

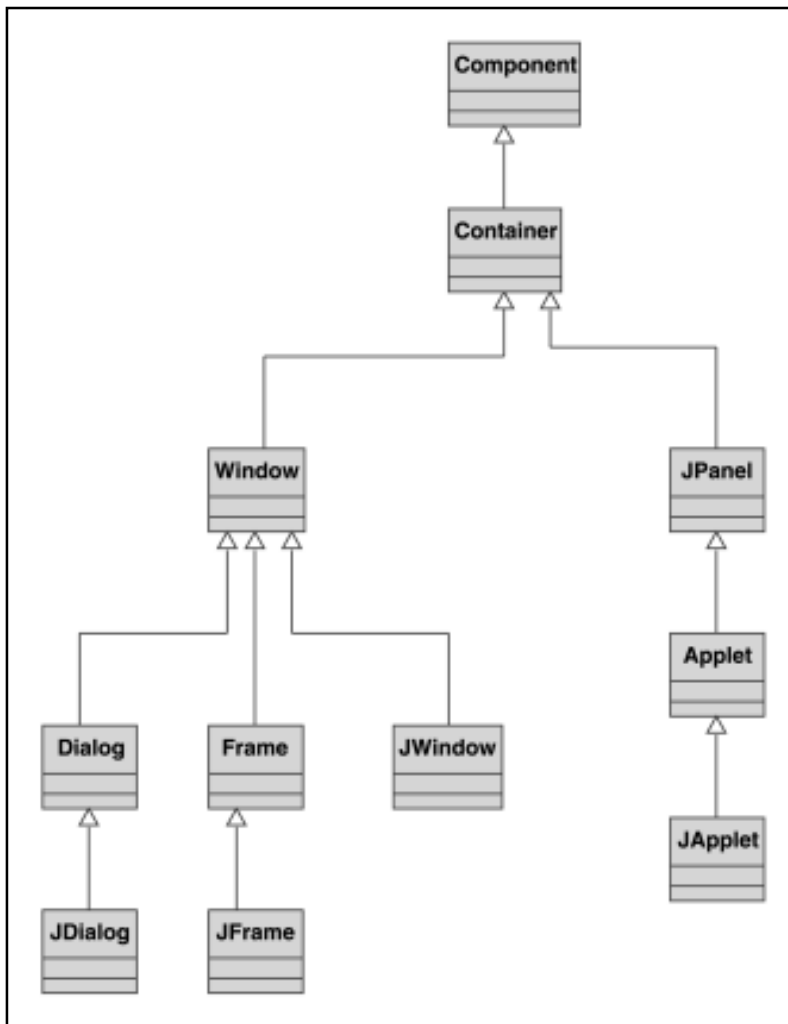
Double Buffering

Il Double Buffering rende il refresh delle interfacce grafiche Swing molto più naturale di quanto avvenisse con i componenti AWT.

I Top Level Container

I Top Level Container sono i componenti all'interno dei quali si creano le interfacce grafiche: ogni programma grafico ne possiede almeno uno, di solito un `JFrame`, che rappresenta la finestra principale.

Come si può vedere dalla gerarchia delle classi, i Top Level Container sono l'interfaccia tra il mondo AWT e quello Swing. Diverse proprietà dei controlli grafici non sono presenti nei Top Level Container: in particolare il cambio di Look & Feel non va a incidere

Figura 10.3 – *I Top Level Container Swing derivano dai corrispondenti componenti AWT*

sull'aspetto delle finestre, che rimangono sempre e comunque uguali alle altre finestre di sistema.

Dal punto di vista implementativo, i Top Level Container presentano una struttura a strati piuttosto complessa, che in questa sede può essere tralasciata. L'unica implicazione degna di nota per l'utente comune è la presenza di un "pannello di contenimento" (Content Pane) accessibile attraverso il metodo `getContentPane()`: questo pannello, e non il `JFrame`, verrà utilizzato come contenitore base per tutti gli altri controlli . Questo dettaglio è l'unica vistosa differenza a cui il programmatore AWT deve abituarsi nel passaggio a Swing. Se su AWT si era abituati a scrivere

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("OK"));
```

ora è necessario scrivere

```
JFrame jf = new JFrame();
Container contentPane = jf.getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(new Button("OK"));
```

o più brevemente

```
JFrame jf = new JFrame();
jf.getContentPane().setLayout(new FlowLayout());
jf.getContentPane().contentPane.add(new Button("OK"));
```

JFrame

È possibile creare una finestra usando i seguenti costruttori

```
JFrame()
```

Crea una finestra.

```
JFrame(String title)
```

Crea una finestra con il titolo specificato dal parametro.

I seguenti metodi permettono di lavorare su alcune proprietà dell'oggetto.

```
void setSize(int width, int height)
```

Specifica la larghezza e l'altezza della finestra.

```
void setResizable(boolean b)
```

Permette di stabilire se si desidera che la finestra sia ridimensionabile.

```
void pack()
```

Ridimensiona la finestra tenendo conto della dimensione ottimale dei componenti posti al suo interno.

```
void setBounds(int x, int y, int width, int height)
```

Oltre alle dimensioni, permette di specificare anche le coordinate dell'angolo in alto a sinistra.

```
void setDefaultCloseOperation(int operation)
```

Imposta l'azione da eseguire alla pressione del tasto `close`. Sono disponibili le seguenti scelte: `WindowConstants.DO_NOTHING_ON_CLOSE` per non ottenere alcun effetto; `WindowConstants.HIDE_ON_CLOSE`, che nasconde la finestra ed è l'impostazione di default; `WindowConstants.DISPOSE_ON_CLOSE` che distrugge la finestra, la quale non potrà più essere aperta; `JFrame.EXIT_ON_CLOSE` (introdotto nel JDK 1.3) che chiude la finestra e termina l'esecuzione del programma.

```
void setTitle(String title)
```

Imposta il titolo della finestra.

```
void setVisible(boolean b)
```

Rende la finestra visibile o invisibile, secondo il valore del parametro.

```
void setJMenuBar(JMenuBar menubar)
```

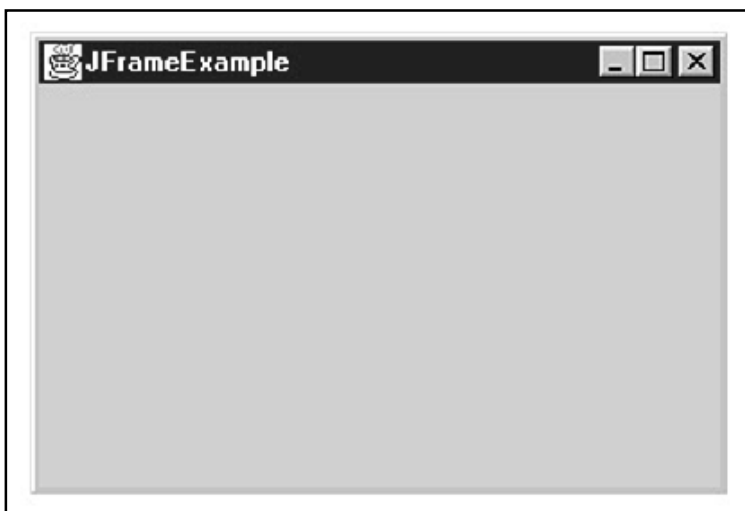
Aggiunge una `JMenuBar` al `Frame`.

```
Container getContentPane()
```

Restituisce il pannello di contenimento del `JFrame`, al cui interno è possibile aggiungere i componenti.

Un breve programma è sufficiente per illustrare come si possa creare un `JFrame`, assegnargli un titolo, una posizione sullo schermo e una dimensione, stabilirne il comportamento in chiusura e renderlo visibile.

```
import javax.swing.*;

public class JframeExample {
   public static void main(String argv[]) {
      JFrame j = new JFrame();
      j.setTitle("JFrameExample");
      j.setBounds(10, 10, 300, 200);
      j.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      j.setVisible(true);
   }
}
```

Figura 10.4 – *Un* `JFrame` *vuoto*

JDialog

Le finestre di dialogo vengono usate per l'inserimento di valori, o per segnalare all'utente una situazione anomala. Ogni finestra di dialogo *appartiene* a un'altra finestra: se si chiude il frame principale, anche i `JDialog` di sua proprietà verranno chiusi. Se si definisce come *modale* un `JDialog`, alla sua comparsa esso *bloccherà* il frame di appartenenza, in modo da costringere l'utente a portare a termine l'interazione. È possibile creare finestre di dialogo con i costruttori seguenti.

```
JDialog(Dialog owner, String title, boolean modal)
```

Crea un `JDialog` modale o non modale, con un titolo e con lo specificato `Dialog` come proprietario.

```
JDialog(Frame owner, String title, boolean modal)
```

Crea un `JDialog` modale o non modale, con un titolo e con lo specificato `Frame` come proprietario.

Altri costruttori permettono di specificare un numero inferiore di parametri. I metodi presentati su `JFrame` sono validi anche su `JDialog`. Ovviamente non è possibile selezionare l'opzione `EXIT_ON_CLOSE` con il metodo `setDefaultCloseOperation()`.

Figura 10.5 – *Il design modulare di Swing permette anche di innestare un controllo all'interno di un altro*

Gerarchia di contenimento

Un'interfaccia grafica è composta da un Top Level Container, ad esempio un `JFrame`, e da un insieme di componenti disposti al suo interno. Swing, grazie al suo design altamente modulare, permette di inserire un qualsiasi componente all'interno di qualunque altro: per esempio è possibile, anche se non particolarmente utile, creare un `JButton` e aggiungere al suo interno un `JComboBox`.

Esistono alcuni componenti che hanno il preciso scopo di fungere da contenitori per altri componenti. Il più usato di questi è senza dubbio `JPanel`, un pannello di uso estremamente generale. Con poche righe si può creare un `JPanel` e inserire un `JButton` al suo interno:

```
JPanel p = new JPanel();
JButton b = new JButton("Button");
p.add(Component c)
```

Più in generale è possibile creare un `JPanel`, disporre alcuni controlli grafici al suo interno, e quindi inserirlo in un altro `JPanel` o nel `ContentPane` di un Top Level Container. Un breve programma permetterà di spiegare meglio i concetti appena illustrati e di introdurre i successivi: si tratta di un esempio abbastanza avanzato, e non c'è da preoccuparsi se alcuni aspetti al principio dovessero risultare poco chiari. Per rendere il programma funzionante è necessario copiare l'immagine `img.gif` nella directory dalla quale si esegue il programma.

```
import javax.swing.*;
import java.awt.*;

public class FirstExample extends JFrame {
   public FirstExample() {
      super("First Example");
      // primo componente
      JTextField textField = new JTextField("Un programma Swing");
      textField.setEditable(false);
      // secondo componente
      JLabel labelIcon = new JLabel(new ImageIcon("img.gif"));
         labelIcon.setBorder(BorderFactory.createLineBorder(Color.black));
      // terzo componente
      JButton okButton = new JButton("OK");
      // Quarto Componente
      JButton cancelButton = new JButton("Cancel");
      // Pannello NORTH
      JPanel northPanel = new JPanel();
      northPanel.setLayout(new GridLayout(1, 0));
      northPanel.setBorder(BorderFactory.createEmptyBorder(10, 4, 10, 4));
      northPanel.add(textField);
      // Pannello CENTER
```

```
        JPanel centralPanel = new JPanel();
        centralPanel.setLayout(new BorderLayout());
        centralPanel.setBorder(BorderFactory.createEmptyBorder(3, 4, 3, 4));
        centralPanel.add(BorderLayout.CENTER,labelIcon);
        // Pannello SOUTH
        JPanel southPanel = new JPanel();
        southPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
        southPanel.add(cancelButton);
        southPanel.add(okButton);
        // Top Level Container
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(BorderLayout.NORTH, northPanel);
        getContentPane().add(BorderLayout.CENTER, centralPanel);
        getContentPane().add(BorderLayout.SOUTH, southPanel);
        pack();
        setVisible(true);
    }

    public static void main(String argv[]) {
        FirstExample fe = new FirstExample();
    }
}
```

Come si può vedere leggendo il sorgente, è stata creata una sottoclasse di `JFrame`; il costruttore assembla l'interfaccia grafica dall'interno all'esterno: dapprima crea i quattro

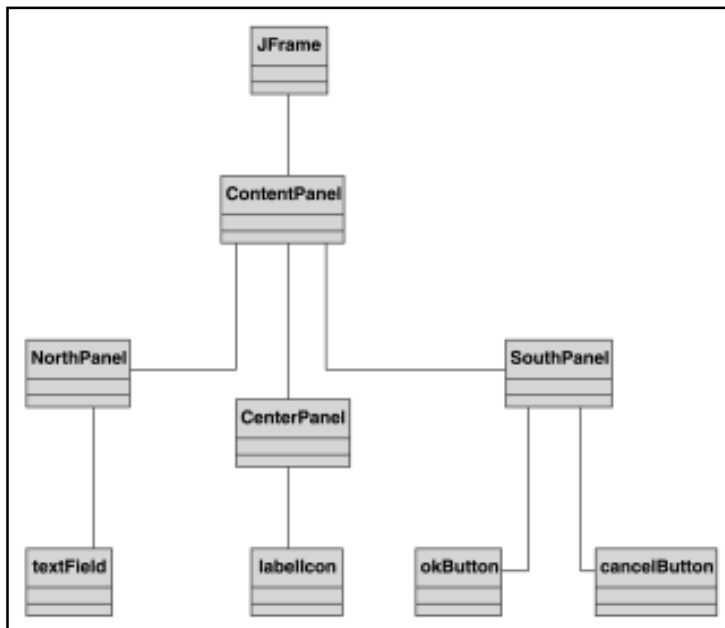
Figura 10.6 – *Un semplice programma di esempio*

componenti più interni, quindi i tre pannelli destinati a contenerli. Su ogni pannello imposta degli attributi, come il bordo o il Layout Manager, che hanno effetto solo all'interno del pannello stesso; infine monta questi pannelli al proprio interno. È possibile rappresentare con un albero la disposizione gerarchica dei componenti di questo programma, come si vede in fig. 10.7.

Progettazione Top Down di interfacce grafiche

Durante la progettazione di interfacce grafiche, può essere utile ricorrere a un approccio Top Down, descrivendo il nostro insieme di componenti a partire dal componente più esterno per poi procedere via via verso quelli più interni. Si può sviluppare una GUI come quella dell'esempio seguendo questo iter:

- Si definisce il tipo di Top Level Container su cui si vuole lavorare, in questo caso un `JFrame`;
- Si assegna un Layout Manager al `JFrame`, in modo da suddividerne la superficie in aree più piccole. Nell'esempio si è ricorsi al `BorderLayout`.

Figura 10.7 – *Gerarchia di contenimento del programma di esempio*

- Per ogni area messa a disposizione dal Layout Manager è possibile definire un nuovo `JPanel`.

- Ogni sottopannello può ricorrere a un Layout Manager differente. Nell'area superiore è stato usato un `GridLayout` per far sì che il `JTextField` occupasse tutta l'area disponibile in larghezza; in quella centrale il `BorderLayout` fa in modo che il disegno sia sempre al centro dell'area disponibile; in basso un `FlowLayout` garantisce che i pulsanti vengano sempre allineati a sinistra. Si noti anche che ogni pannello definisce un proprio bordo.

- Ogni pannello identificato nel terzo passaggio potrà essere sviluppato ulteriormente, creando al suo interno ulteriori pannelli, o disponendo dei controlli. Nell'esempio è stato aggiunto al primo pannello un `JTextField`, nel secondo una `JLabel` contenente un'icona e nel terzo due `JButton`.

Terminata la fase progettuale, si può passare a scrivere il codice per i nostri controlli. In questa seconda fase adotteremo l'approccio Bottom Up, cioè scriveremo per primo il codice dei componenti atomici, quindi quello dei contenitori e infine quello del `JFrame`.

La gestione degli eventi

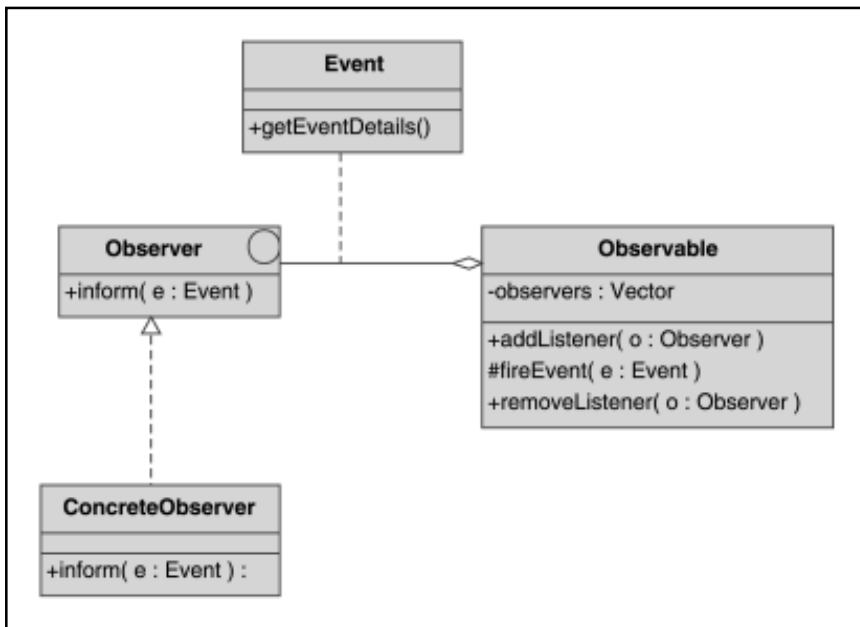
Il modello a eventi di Swing

La gestione degli eventi su Swing è equivalente a quella introdotta su AWT con il JDK 1.1. Ogni oggetto grafico è predisposto a essere sollecitato in qualche modo dall'utente: un pulsante può essere premuto, una finestra può essere chiusa, il cursore del mouse può essere spostato e così via. Queste interazioni generano degli eventi che vengono gestiti secondo una modalità definita in letteratura *Event Delegation*, la quale prevede che un componente grafico *deleghi* a un *ascoltatore* l'azione da svolgere. Un pulsante, per così dire, *non sa* cosa avverrà alla sua pressione: esso si limita a *notificare* ai propri ascoltatori che l'evento che essi attendevano è avvenuto, e questi provvederanno a produrre l'effetto desiderato.

Si noti che ogni componente grafico può avere più di un ascoltatore per un determinato evento: in questo caso essi saranno chiamati uno per volta secondo l'ordine in cui si sono registrati. Se l'utente cerca di scatenare un nuovo evento prima che il precedente sia stato consumato (ad esempio premendo ripetutamente un pulsante), ogni nuovo evento verrà bufferizzato, e l'azione corrispondente sarà eseguita solo al termine della precedente.

Il pattern Observer

Il modello *Event Driven* è una implementazione del pattern Observer, lo schema di

Figura 10.8 – *Il pattern Observer*

progettazione descritto dal diagramma in fig. 10.8. C'è un oggetto in grado di scatenare un determinato tipo di evento, a cui sarà dato il nome `Observable` per sottolineare la sua caratteristica di poter essere *osservato*. Esso mantiene una lista di *ascoltatori*, definiti dall'interfaccia `Observer`, all'interno di un vettore `observers` un ascoltatore può registrarsi presso `Observable` usando il metodo `addListener()`, mentre si esclude dalla lista degli ascoltatori chiamando `removeListener()`. Ogni volta che si verifica un evento, `Observable` crea un'istanza di `Event`, un oggetto che racchiude i dettagli dell'evento, e lo invia a tutti gli ascoltatori utilizzando `fireEvent(Event e)`, un metodo che esegue una `inform(Event e)` su tutti gli ascoltatori presenti nel Vector `observers`. Nelle seguenti righe vediamo una possibile implementazione della classe Observable, non molto diversa da quella effettivamente utilizzata nel codice dei componenti Swing. Si noti che questo breve programma presenta l'implementazione standard dei metodi di servizio di una sorgente di eventi conforme al modello Observable, mentre non illustra il codice che genera l'evento, dal momento che tale codice può variare enormemente da caso a caso.

```
public class Observable {
private Vector observers = new Vector();
public void addListener(Observer o) {
   observers.add(o);
```

```
}
public void removeListener(Observer o) {
   observers.remove(o);
}
protected void fireEvent(Event e) {
   for(int i=0; i<observers.size(); i++) {
      Observer observer = (Observer)observers.elementAt(i);
      observer.inform(e);
   }
}
}
```

L'ascoltatore viene definito come un'interfaccia Java; questa scelta pone al programmatore un vincolo riguardo a *come* scrivere un ascoltatore, ma lascia la più completa libertà su *cosa* l'ascoltatore debba fare. Si può scrivere una classe che stampa messaggi sullo schermo, un'altra che fa comparire una finestra di dialogo, una che colora lo schermo di bianco e una che termina il programma; si può addirittura scrivere una classe che non fa niente: se queste classi implementano l'interfaccia `Observer` e descrivono il proprio comportamento all'interno del metodo `inform(Event e)`, esse risulteranno ascoltatori validi per l'oggetto `Observable`. In fig. 10.8 si vede la classe `ConcreteObserver`, una possibile implementazione dell'interfaccia `Observer`.

Di seguito sarà affrontato lo studio graduale dei più importanti controlli grafici; per ogni controllo, verrà introdotto anche il rispettivo ascoltatore.

I controlli di base

Bottoni

I pulsanti, grazie alla loro modalità di utilizzo estremamente intuitiva, sono sicuramente i controlli grafici più usati. Swing offre quattro tipi di pulsanti, legati tra loro dalla gerarchia illustrata in fig. 10.9.

`JButton` è l'implementazione del comune bottone push; `JToggleButton` è un pulsante on/off; `JCheckBox` è un controllo grafico creato sul modello delle caselle di spunta dei questionari; `JRadioButton` è un controllo che permette di scegliere una tra molte possibilità in modo mutuamente esclusivo.

La classe `AbstractButton` definisce l'interfaccia di programmazione comune a tutti i pulsanti: per questa ragione si comincerà lo studio da questa classe. Successivamente verranno introdotte le sottoclassi concrete.

`AbstractButton`: gestione dell'aspetto

L'API di `AbstractButton` definisce un insieme di metodi per gestire l'aspetto del

Figura 10.9 – *Gerarchia dei bottoni Swing*

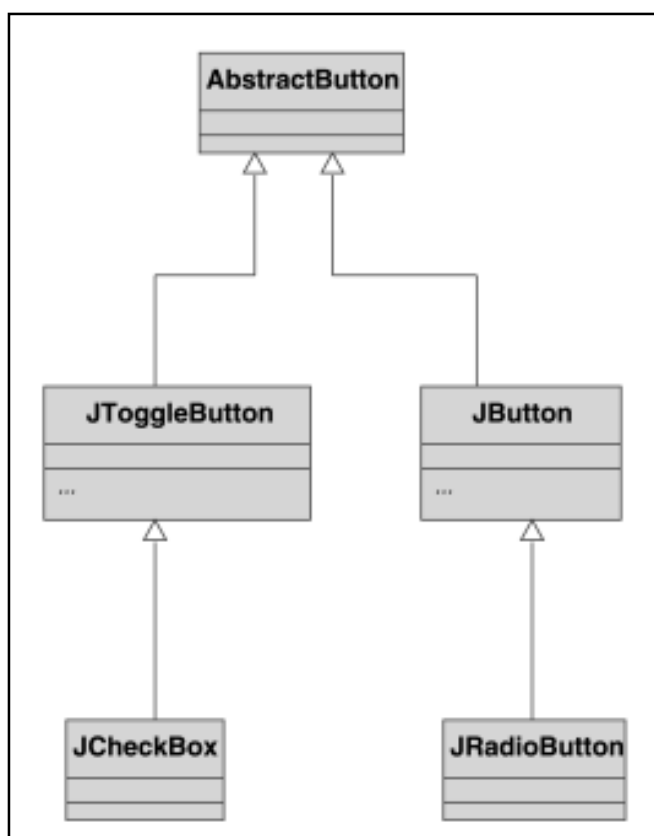



Figura 10.10 – *Pulsanti disponibili su Swing*

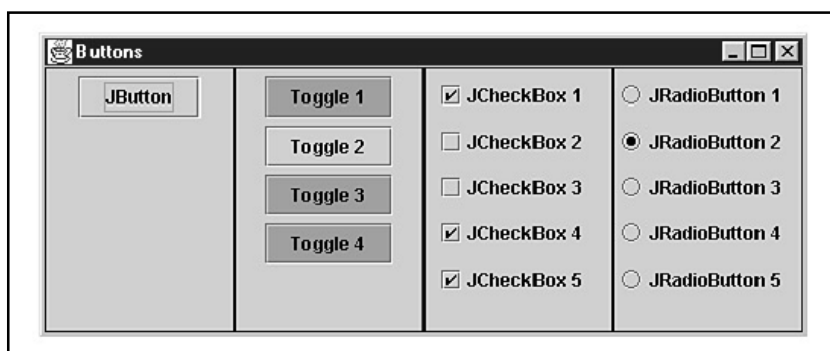



Figura 10.11 – *È possibile decorare un pulsante con HTML*

componente; in particolare viene fornita la possibilità di associare ad ogni controllo grafico un'etichetta di testo, un'icona o entrambi. È possibile impostare l'etichetta in formato HTML: basta aggiungere il prefisso `<html>` nel parametro di `setText(String)`. Le seguenti righe di codice mostrano come creare un `JButton` con un'etichetta HTML:

```
JButton b = new JButton();
b.setText("<html><font size=-1><b><u>Esempio</u></b> di pulsante <b>HTML</b></font></html>"));
```

I pulsanti Swing permettono di impostare un'icona diversa per ognuno degli stati in cui si possono trovare: normale, premuto, selezionato (valido per i controlli che mantengono lo stato come i CheckBox), disabilitato e rollover (lo stato in cui si trova un controllo quando viene sorvolato dal puntatore del mouse).

```
void setText(String text)
```

Imposta l'etichetta di testo.

```
void setEnabled(boolean b)
```

Abilita o disabilita il controllo.

```
void setRolloverEnabled(boolean b)
```

Attiva o disattiva l'effetto rollover.

```
void setSelected(boolean b)
```

Regola lo stato del controllo.

```
void setIcon(Icon defaultIcon)
```

Imposta l'icona di default.

```
void setPressedIcon(Icon pressedIcon)
```

Imposta l'icona per lo stato "premuto".

```
void setSelectedIcon(Icon selectedIcon)
```

Imposta l'icona "selezionato".

```
void setDisabledIcon(Icon disabledIcon)
```

Imposta l'icona "disabilitato".

```
void setRolloverIcon(Icon rolloverIcon)
```

Imposta l'icona "rollover".

```
void setDisabledSelectedIcon(Icon disabledSelectedIcon)
```

Imposta l'icona per la combinazione "disabilitato – selezionato".

```
void setRolloverSelectedIcon(Icon rolloverSelectedIcon)
```

Imposta l'icona per la combinazione "rollover selezionato".

Come icona si può usare un oggetto di tipo `ImageIcon`, ricorrendo al costruttore `ImageIcon(String filename)` che crea un'icona a partire da un'immagine di tipo GIF o JPEG il cui percorso viene specificato nella stringa del parametro.

Se non viene specificato diversamente, le immagini per gli stati "premuto", "selezionato" e "disabilitato" vengono create in modo automatico a partire da quella di default. Le seguenti righe creano un pulsante decorato con l'immagine `img.gif` (può andar bene una qualunque immagine GIF o JPEG): ovviamente il file deve essere presente nella directory di lavoro, altrimenti verrà creato un pulsante vuoto.

```
// crea un pulsante
JButton b = new JButton();
// crea un'icona
ImageIcon icon = new ImageIcon("img.gif");
// imposta icon come icona per b
b.setIcon(icon);
```

`AbstractButton`: gestione degli eventi

I controlli derivati da `AbstractButton` prevedono due tipi di ascoltatore: `ActionListener` e `ItemListener`. Il primo ascolta il tipo più semplice di evento, come la pressione di un pulsante; il secondo permette di conoscere i cambiamenti da "selezionato" a "non selezionato" o viceversa nei pulsanti tipo `JToggleButton`. Nei paragrafi seguenti si vedranno alcuni esempi di utilizzo; qui di seguito è possibile vedere i metodi necessari ad aggiungere o rimuovere gli ascoltatori da un pulsante.

```
void addActionListener(ActionListener l)
```

Aggiunge un `ActionListener` al pulsante.

```
void removeActionListener(ActionListener l)
```

Rimuove un `ActionListener` dal pulsante.

```
void addItemListener(ItemListener l)
```

Aggiunge un `ItemListener` al pulsante.

```
void removeItemListener(ItemListener l)
```

Rimuove un `ItemListener` dal pulsante.

È da segnalare la possibilità di attivare le azioni associate a un pulsante in maniera equivalente a quella che si otterrebbe con un click del mouse.

```
void doClick()
```

Esegue un click sul pulsante.

JButton

Ecco ora la prima sottoclasse concreta di `AbstractButton`: `JButton`, il comune pulsante push. I seguenti costruttori permettono di creare pulsanti e di definirne le principali proprietà.

```
JButton(String text)
```

Crea un pulsante con l'etichetta specificata dal parametro.

```
JButton(Icon icon)
```

Crea un pulsante con l'icona specificata dal parametro.

```
JButton(String text, Icon icon)
```

Crea un pulsante con l'etichetta e l'icona specificate dai parametri.

Ogni Top Level container può segnalare un pulsante come `DefaultButton`. Esso verrà evidenziato in maniera particolare e verrà richiamato con la semplice pressione del tasto "Invio". Le righe seguenti creano un pulsante, un `JFrame` e impostano il pulsante come `DefaultButton`.

```
JFrame f = new JFrame();
JButton b = new JButton("DefaultButton");
f.getContentPane().add(b);
f.getRootPane().setDefaultButton(b);
```

L'ascoltatore più utile per un `JButton` è `ActionListener`, che riceve eventi di tipo `ActionEvent` quando il pulsante viene premuto e rilasciato. Di seguito ecco i metodi più importanti di `ActionEvent`.

Figura 10.12 – *Relazione tra* `JButton` *e* `ActionEvent`*: si noti la conformità al pattern Observer descritto in fig. 10.8*

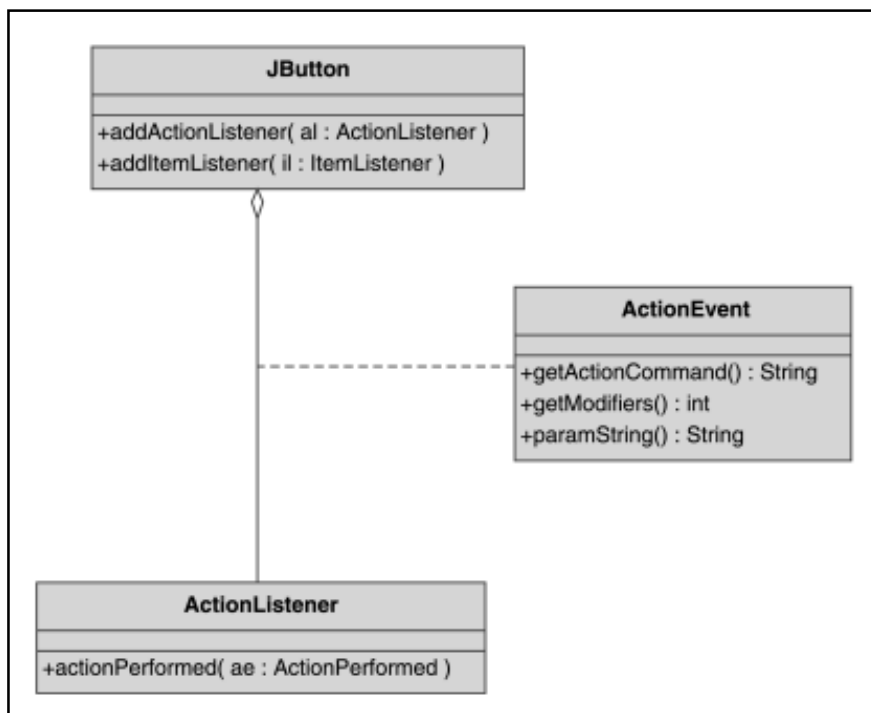



Figura 10.13 – *Esempio di uso di* `JButton`

```
String getActionCommand()
```

Restituisce una stringa che identifica il comando (se non specificata altrimenti, è uguale all'etichetta del pulsante).

```
String paramString()
```

Restituisce un parametro che identifica l'evento.

Si vedrà ora un programma di esempio che illustra l'uso di due `JButton`, uno dei quali viene registrato come `DefaultButton`, e dei relativi ascoltatori.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JButtonExample extends JFrame {
   private JDialog dialog;
   private JButton okButton;
   private JButton jDialogButton;

   public JButtonExample() {
      // Imposta le proprietà del Top Level Container
      super("JButtonExample");
      setBounds(10, 35, 200, 70);
      getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

      // Crea una finestra di dialogo modale,
      // inizialmente invisibile
      dialog = new JDialog(this, "JDialog", true);
      dialog.setBounds(250, 20, 300, 100);
      dialog.getContentPane().setLayout(new BorderLayout());
      dialog.getContentPane().add(BorderLayout.CENTER,
                              new JLabel("Chiudi questa finestra per continuare",
                              JLabel.CENTER));
      dialog.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);

      // Crea due pulsanti
      okButton = new JButton("OK");
      jDialogButton = new JButton("Open Frame");
      // Crea gli ascoltatori
      ActionListener okListener = new OKButtonListener();
      ActionListener openActionListener = new JDialogButtonListener();
      // Registra gli ascoltatori presso i pulsanti
      okButton.addActionListener(okListener);
```

```
      jDialogButton.addActionListener(openActionListener);

      //imposta okButton come DefaultButton
      getRootPane().setDefaultButton(okButton);

      // Aggiunge i pulsanti al Top Level Container
      getContentPane().add(okButton);
      getContentPane().add(jDialogButton);

      setVisible(true);
   }
   // Ascoltatore del pulsante OK
   class OKButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         try {
            System.exit(0);
         }
         catch (Exception ex) { }
      }
   }
   // Ascoltatore del pulsante jDialog
   class JDialogButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         dialog.setVisible(true);
      }
   }
   public static void main(String argv[]) {
      JButtonExample b = new JButtonExample();
   }
}
```

`JToggleButton`

È un pulsante che può avere due stati: premuto e rilasciato. Cliccando con il mouse si provoca il passaggio tra uno stato e l'altro. Alcuni costruttori permettono di creare `JToggleButton` e di impostarne le proprietà.

```
JToggleButton(String text, Icon icon, boolean selected)
```

Crea un pulsante con l'etichetta, l'icona e lo stato specificate dai parametri.

```
JToggleButton(String text, boolean selected)
```

Crea un pulsante con l'etichetta e lo stato specificate dai parametri

Figura 10.14 – *Relazione Observer–Observable tra* `JToggleButton` *e* `ItemEvent`

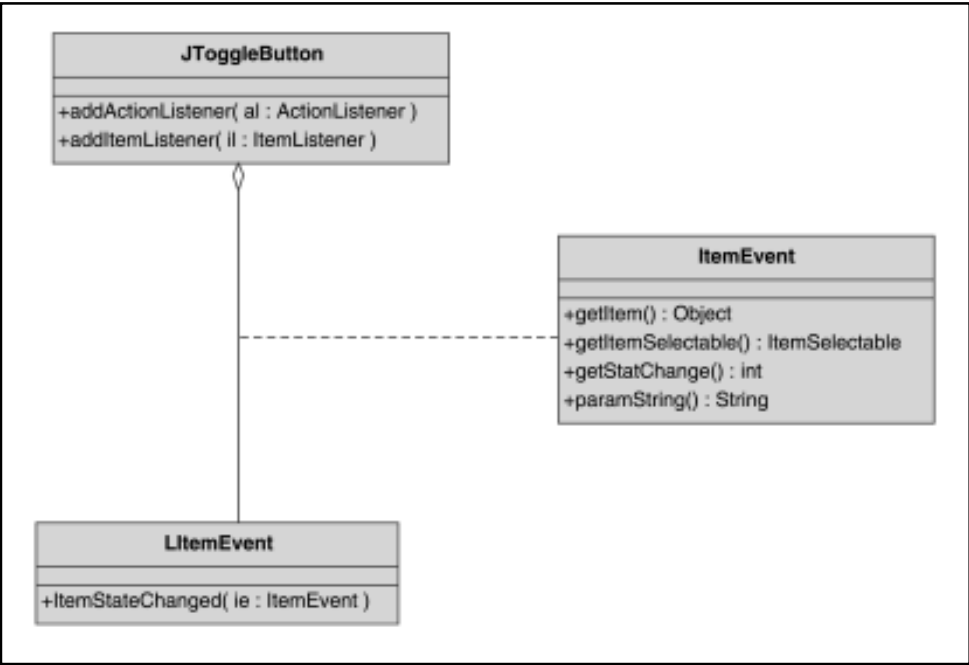



Figura 10.15 – *Esempio di* `JToggleButton`

`JToggleButton(Icon icon, boolean selected)`

Crea un pulsante con l'icona e lo stato specificati dai parametri, senza etichetta.

Un `JToggleButton`, quando viene premuto, genera un `ActionEvent` e un `ItemEvent`. L'evento più interessante per questo tipo di pulsanti è il secondo, che segnala il *cambiamento di stato*, cioè il passaggio da premuto a rilasciato e viceversa.

`Object getItem()`

Restituisce l'oggetto che ha generato l'evento.

`int getStateChange()`

Restituisce un intero il quale può assumere i due valori `ItemEvent.SELECTED` oppure `ItemEvent.DESELECTED`.

`String paramString()`

Restituisce un parametro che identifica l'evento.

È possibile comunque utilizzare ascoltatori di tipo `ActionListener` nella stessa maniera illustrata nell'esempio precedente: in questo caso verrà inviato un `ActionEvent` ad ogni click, indipendentemente dallo stato del pulsante. L'esempio che seguirà crea una finestra con un `JToggleButton` che permette di aprire e di chiudere una finestra di dialogo non modale.

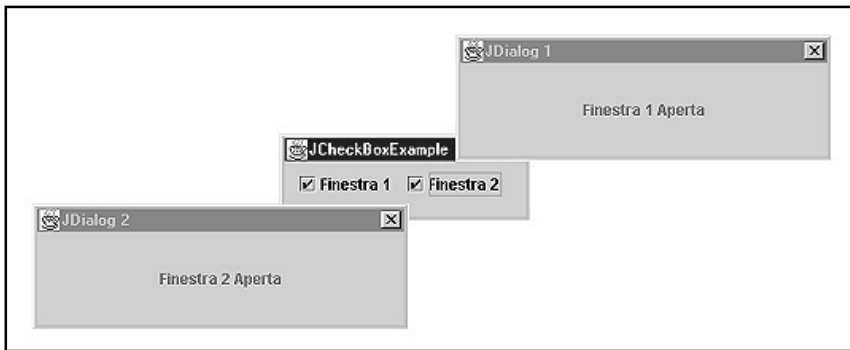
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JToggleButtonExample extends JFrame {

   private JDialog dialog;
   private JToggleButton jDialogButton;

   public JToggleButtonExample() {
      // Imposta le proprietà del Top Level Container
      super("JToggleButtonExample");
      setBounds(10, 35, 250, 70);
```

```
      getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

      // Crea una finestra di dialogo non modale,
      // inizialmente invisibile
      dialog = new JDialog(this, "JDialog", false);
      dialog.setBounds(250, 20, 300, 100);
      dialog.getContentPane().setLayout(new BorderLayout());
      dialog.getContentPane().add(BorderLayout.CENTER,
                                  new JLabel("Finestra Aperta", JLabel.CENTER));
      dialog.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

      // Crea il pulsante e lo registra presso il suo ascoltatore
      jDialogButton = new JToggleButton("Open / Close Frame", false);
      jDialogButton.addItemListener(new JDialogButtonItemListener());

      // Aggiunge il pulsante al Top Level Container
      getContentPane().add(jDialogButton);
      setVisible(true);
   }
   // Ascoltatore di JDialogButton
   class JDialogButtonItemListener implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
         int status = e.getStateChange();
         if(status == ItemEvent.SELECTED)
            dialog.setVisible(true);
         else
            dialog.setVisible(false);
         }
      }
   public static void main(String argv[]) {
      JToggleButtonExample b = new JToggleButtonExample();
   }
}
```

Il codice dell'ascoltatore `JDialogButtonItemListener` è un po' più complesso di quello degli `ActionListener` dell'esempio precedente: questo tipo di ascoltatore deve normalmente prevedere una verifica dello stato in cui si trova il pulsante, al fine di produrre la reazione appropriata. La verifica dello stato viene effettuata interrogando l'oggetto `ItemEvent` con il metodo `getStateChange()`.

JCheckBox

`JCheckBox` è una sottoclasse di `JToggleButton` che ha una forma simile alle caselle di spunta dei questionari. Il suo funzionamento è analogo a quello della superclasse, ma di fatto tende a essere utilizzato in contesti in cui si offre all'utente la possibilità di sceglie-

Figura 10.16 – *Con i due* `JCheckBox` *è possibile selezionare due finestre di dialogo*

re una o più opzioni tra un insieme, come avviene ad esempio nei pannelli di controllo. I costruttori disponibili sono gli stessi di `JToggleButton`, e così pure la gestione degli eventi, pertanto non sarà necessario ripeterli. Un esempio mostrerà un uso tipico di questo componente.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JCheckBoxExample extends JFrame {

   private JDialog dialog1;
   private JDialog dialog2;
   private JCheckBox checkBox1;
   private JCheckBox checkBox2;

   public JCheckBoxExample() {
      // Imposta le proprietà del Top Level Container
      super("JCheckBoxExample");
      setBounds(10, 35, 200, 70);
      getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

      // Crea due finestre di dialogo non modali,
      // inizialmente invisibili
      dialog1 = new JDialog(this, "JDialog 1", false);
      dialog1.setBounds(250, 20, 300, 100);
      dialog1.getContentPane().setLayout(new BorderLayout());
      dialog1.getContentPane().add(BorderLayout.CENTER,
                                   new JLabel("Finestra 1 Aperta", JLabel.CENTER));
      dialog1.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
      dialog2 = new JDialog(this, "JDialog 2", false);
```

```
      dialog2.setBounds(250, 150, 300, 100);
      dialog2.getContentPane().setLayout(new BorderLayout());
      dialog2.getContentPane().add(BorderLayout.CENTER,
                           new JLabel("Finestra 2 Aperta", JLabel.CENTER));
      dialog2.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

      // Crea i checkBox e li registra presso il loro ascoltatore
      ItemListener listener = new JCheckBoxItemListener();
      checkBox1 = new JCheckBox("Finestra 1");
      checkBox1.addItemListener(listener);
      checkBox2 = new JCheckBox("Finestra 2");
      checkBox2.addItemListener(listener);

      // Aggiunge i checkBox al Top Level Container
      getContentPane().add(checkBox1);
      getContentPane().add(checkBox2);
      setVisible(true);
   }
   // ascoltatore JCheckBox
   class JCheckBoxItemListener implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
         Object target = e.getItem();
         int status = e.getStateChange();

         if(target.equals(checkBox1) && status == ItemEvent.SELECTED)
            dialog1.setVisible(true);
         else if(target.equals(checkBox1) && status == ItemEvent.DESELECTED)
            dialog1.setVisible(false);
         else if(target.equals(checkBox2) && status == ItemEvent.SELECTED)
            dialog2.setVisible(true);
         else if(target.equals(checkBox2) && status == ItemEvent.DESELECTED)
            dialog2.setVisible(false);
      }
   }
   public static void main(String argv[]) {
      JCheckBoxExample b = new JCheckBoxExample();
   }
}
```

L'ascoltatore `JCheckBoxItemListener` presenta un grado di complessità maggiore del precedente: esso ascolta entrambi i controlli e ad ogni chiamata verifica quale dei due abbia generato l'evento (chiamando il metodo `getItem()` di `ItemEvent`) e quale stato ha assunto, predisponendo quindi la reazione opportuna.

JRadioButton

`JRadioButton` è una sottoclasse di `JToggleButton`, dotata dei medesimi costruttori.

Questo tipo di controlli viene usato tipicamente per fornire all'utente la possibilità di operare una scelta tra un insieme di possibilità, in contesti nei quali una scelta esclude l'altra. Per implementare questo comportamento di mutua esclusione è necessario registrare i `JRadioButton` che costituiscono l'insieme presso un'istanza della classe `ButtonGroup`, come viene mostrato nelle righe seguenti.

```
ButtonGroup group = new ButtonGroup();
group.add(radioButton1);
group.add(radioButton2);
group.add(radioButton3);
```

Ogni volta che l'utente attiva uno dei pulsanti registrati presso il `ButtonGroup`, gli altri vengono automaticamente messi a riposo. Questo comportamento ha una conseguenza importante nella gestione degli eventi: un gruppo di `JRadioButton` registrati presso un `ButtonGroup` genera *due* `ItemEvent` consecutivi per ogni click del mouse, uno per la casella che viene selezionata e uno per quella deselezionata.

Di norma si è interessati unicamente al fatto che un particolare `JRadioButton` sia stato premuto, poiché la politica di mutua esclusione rende superfluo verificarne lo stato: in questi casi è consigliabile utilizzare un `ActionListener` come nell'esempio seguente, nel quale un gruppo di `JRadioButton` permette di modificare la scritta su un'etichetta di testo.

```
import javax.swing.*;
import java.awt.*;
```

Figura 10.17 – *Attivando un JRadioButton l'etichetta viene modificata*

```
import java.awt.event.*;

public class JRadiobuttonExample extends JFrame {

   private JRadioButton radioButton1;
   private JRadioButton radioButton2;
   private JRadioButton radioButton3;
   private JLabel label;

   public JRadiobuttonExample() {
      // Imposta le proprietà del Top Level Container
      super("JRadiobuttonExample");
      setBounds(10, 35, 150, 150);
      getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

      // Crea i radiobutton e la label
      radioButton1 = new JRadioButton("RadioButton 1");
      radioButton2 = new JRadioButton("RadioButton 2");
      radioButton3 = new JRadioButton("RadioButton 3");
      label = new JLabel();

      // Crea l'ascoltatore e registra i JRadioButton
      ActionListener listener = new JRadioButtonListener();
      radioButton1.addActionListener(listener);
      radioButton2.addActionListener(listener);
      radioButton3.addActionListener(listener);

      // Crea il ButtonGroup e registra i RadioButton
      ButtonGroup group = new ButtonGroup();
      group.add(radioButton1);
      group.add(radioButton2);
      group.add(radioButton3);

      // Aggiunge i componenti al Top Level Container
      getContentPane().add(radioButton1);
      getContentPane().add(radioButton2);
      getContentPane().add(radioButton3);
      getContentPane().add(label);

      radioButton1.doClick();
      setVisible(true);
   }
   // Ascoltatore JRadioButton
   class JRadioButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         String target = e.getActionCommand();
         label.setText(target);
      }
   }
```

```
   public static void main(String argv[]) {
      JRadiobuttonExample b = new JRadiobuttonExample();
   }
}
```

JToolBar

Nelle moderne interfacce grafiche l'insieme dei controlli viene suddiviso principalmente tra due luoghi: la Menu Bar, di cui si parlerà più avanti, e la Tool Bar, di cui ci si occupa ora.

`JToolBar` è un contenitore che permette di raggruppare un insieme di controlli grafici in una riga che, nella maggioranza dei casi, viene posizionata al di sotto della barra dei menu. Sebbene sia utilizzata principalmente come contenitore di pulsanti provvisti di icona, è possibile inserire al suo interno qualunque tipo di componente, come campi di testo o liste di selezione a discesa.

Ricorrendo al Drag & Drop è possibile "staccare" una Tool Bar dalla sua posizione originale e renderla *fluttuante*: in questo caso essa verrà visualizzata in una piccola finestra separata dal Frame principale. Allo stesso modo è possibile "afferrare" una Tool Bar con il mouse e trascinarla in una nuova posizione: questa possibilità di personalizzazione è fornita dalla maggior parte dei programmi grafici.

Usare `JToolBar` all'interno dei propri programmi non presenta particolari difficoltà: è sufficiente crearla, aggiungervi i componenti nell'ordine da sinistra a destra, e posizionarla all'interno del contenitore principale. Più avanti verranno presentati dei programmi che ne fanno uso.

```
JToolBar toolBar = new JToolBar();
ImageIcon icon = new ImageIcon("img.gif");
JButton b = new JButton(icon);
toolBar.add(b);
...
JFrame f = new JFrame();
f.getContentPane().setLayout(new BorderLayout());
f.getContentPane().add(BorderLayout.NORTH, toolBar);
```

Di seguito viene presentato un riassunto dell'API `JToolBar`.

```
public JToolBar()
```

Crea una Tool Bar.

```
public Component add(Component c)
```

Aggiunge un componente alla Tool Bar. L'ordine di immissione dei componenti è da sinistra a destra.

```
public JButton add(Action a)
```

Crea un pulsante corrispondente alla `Action` del parametro e lo aggiunge alla Tool Bar.

```
public void addSeparator()
```

Aggiunge un separatore alla Tool Bar.

```
public void setFloatable(boolean b)
```

Se si passa `true` come parametro, la Tool Bar viene impostata come removibile, quindi può essere "staccata" dalla sua posizione originale e trascinata in una nuova posizione.

I menu

I menu sono controlli che permettono di accedere a un grande numero di opzioni in uno spazio ridotto, organizzato gerarchicamente.Ogni programma grafico dispone di una Menu Bar organizzata per gruppi di funzioni: accesso al disco, operazioni sulla clipboard, opzioni e così via; ogni menu può essere composto da elementi attivi (`MenuItem`) o da ulteriori menu nidificati.

In Swing anche i menu si assemblano in maniera gerarchica, costruendo un oggetto per ogni elemento e aggiungendolo al proprio contenitore. La gerarchia delle classi in fig. 10.18 mostra che ogni sottoclasse di `JComponent` è predisposta a contenere menu, capacità che viene garantita dall'interfaccia `MenuContainer`; le classi `JMenu` e `JPopupMenu` sono contenitori appositamente realizzati per questo scopo. La classe `JMenuItem` implementa l'elemento di tipo più semplice: essendo sottoclasse di `AbstractButton`, ne eredita l'interfaccia di programmazione e il comportamento (vale a dire che tutti i metodi visti nel paragrafo *AbstractButton* possono essere utilizzati anche su questi componenti). `JRadioButtonMenuItem` e `JCheckBoxMenuItem` sono Menu Items analoghi ai pulsanti `JRadioButton` e `JCheckBox`; oltre alla parentela diretta con `JMenuItem`, essi hanno in comune con `JMenu` e `JPopupMenu` l'interfaccia `MenuElement`, che accomuna tutti i componenti che possono comparire all'interno di un menu.

Figura 10.18 – *Gerarchia di* `JMenu`

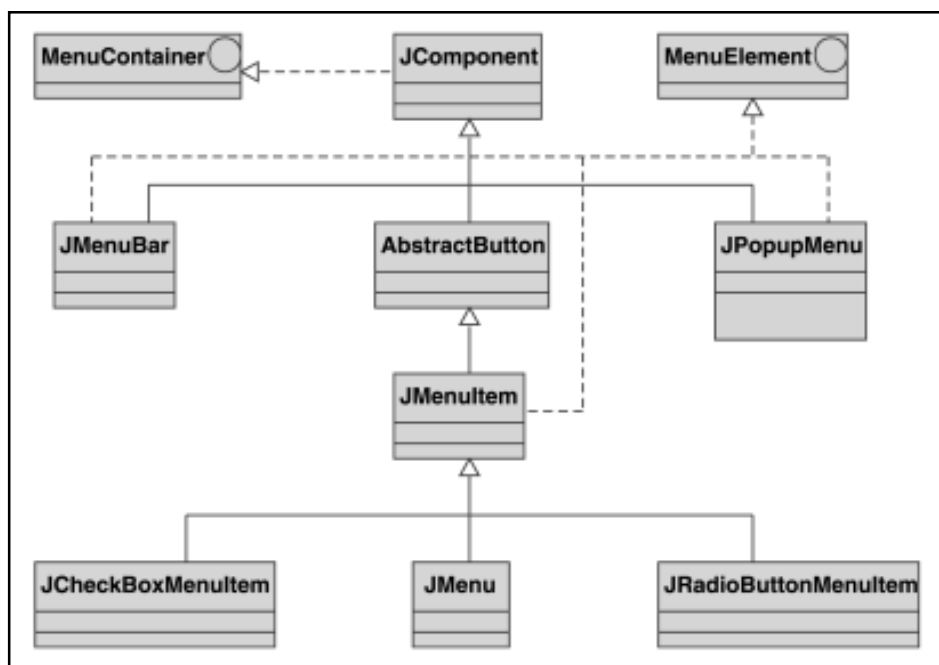
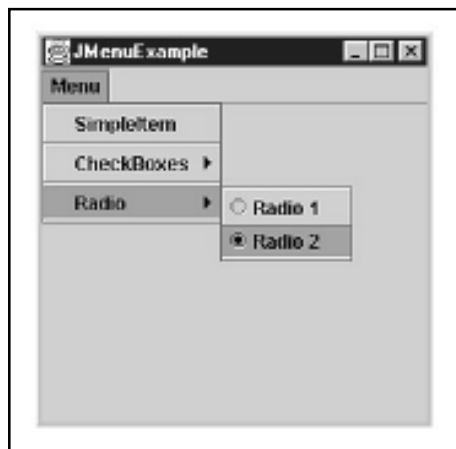



Figura 10.19 – *Esempio di Menu*

Ecco costruttori e metodi per le classi JMenuBar e JMenu.

```
JMenuBar()
```

Crea una JMenuBar.

```
JMenu(String text)
```

Crea un JMenu con l'etichetta specificata dal parametro.

Alcuni metodi sono comuni a entrambe le classi.

```
void add(JMenu m)
```

Aggiunge un JMenu.

```
void remove(JMenu m)
```

Rimuove un JMenu.

```
void removeAll()
```

Rimuove tutti i JMenu.

I seguenti costruttori permettono di creare JMenuItem, JRadioButtonMenuItem e JCheckBoxMenuItem in maniera simile a come si può fare con i JButton. I parametri permettono di specificare l'etichetta, l'icona e lo stato.

```
JMenuItem(String text)
JMenuItem(String text, Icon icon)
JCheckBoxMenuItem(String text, Icon icon, boolean b)
JCheckBoxMenuItem(String text, boolean b)
JRadioButtonMenuItem(String text, boolean selected)
JRadioButtonMenuItem(String text, Icon icon, boolean selected)
```

Sebbene sia possibile posizionare una JMenuBar ovunque all'interno di un'interfaccia grafica, i Top Level Container JFrame, JApplet e JDialog riservano a questo scopo

un posto esclusivo, situato appena sotto alla barra del titolo. È possibile aggiungere un `JMenu` a un `JFrame`, a un `JApplet` o a un `JDialog` usando il metodo `setJMenuBar(JMenuBar)`, come si vede nelle righe di esempio.

```
JMenuBar menubar = new JMenuBar();
...
JFrame f = new JFrame("A Frame");
f.setJMenuBar(menuBar)
```

Il seguente esempio illustra la costruzione di un menu ricorrendo a elementi di ogni tipo.

```
import javax.swing.*;

public class JMenuExample extends JFrame {
   public JMenuExample() {
      // Imposta le proprietà del Top Level Container
      super("JMenuExample");
      setBounds(10, 35, 250, 250);
      // Crea menu, sottomenu e menuitems
      JMenuBar menubar = new JMenuBar();
      JMenu menu = new JMenu("Menu");
         JMenuItem simpleItem = new JMenuItem("SimpleItem");
         JMenu checkSubMenu = new JMenu("CheckBoxes");
            JCheckBoxMenuItem check1 = new JCheckBoxMenuItem("Check 1");
            JCheckBoxMenuItem check2 = new JCheckBoxMenuItem("Check 1");
         JMenu radioSubMenu = new JMenu("Radio");
            JRadioButtonMenuItem radio1 = new JRadioButtonMenuItem("Radio 1");
            JRadioButtonMenuItem radio2 = new JRadioButtonMenuItem("Radio 2");
            ButtonGroup group = new ButtonGroup();
            group.add(radio1);
            group.add(radio2);
            // Compone i menu
            checkSubMenu.add(check1);
            checkSubMenu.add(check2);
            radioSubMenu.add(radio1);
            radioSubMenu.add(radio2);
         menu.add(simpleItem);
         menu.addSeparator();// (new JSeparator());
         menu.add(checkSubMenu);
         menu.addSeparator();//.add(new JSeparator());
         menu.add(radioSubMenu);
      menubar.add(menu);
      // Aggiunge la menubar al JFrame
      setJMenuBar(menubar);
      setVisible(true);
   }
   public static void main(String argv[]) {
```

```
      JMenuExample m = new JMenuExample();
   }
}
```

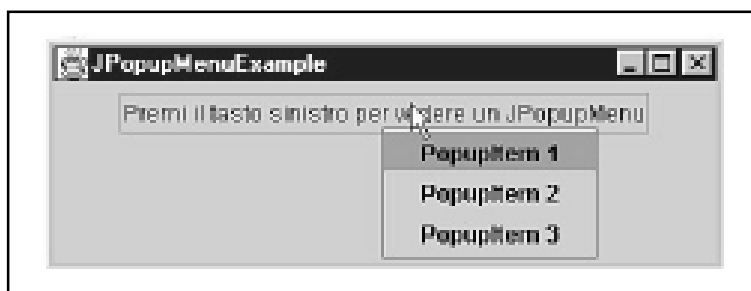
JPopupMenu

I `JPopupMenu` implementano i menu contestuali presenti in quasi tutti i moderni sistemi a finestre. La costruzione di `JPopupMenu` è del tutto simile a quella di `JMenu`; mentre diversa è la modalità di visualizzazione. Il metodo

```
public void show(Component invoker, int x, int y)
```

visualizza il menu al di sopra del componente specificato dal parametro `invoker`, alle coordinate `x` e `y` (relative a `invoker`). Per associare un `JPopupMenu` alla pressione del tasto destro del mouse su un oggetto grafico, è necessario registrare il componente interessato presso un `MouseListener` incaricato di chiamare il metodo `show()` al momento opportuno. Dal momento che alcuni sistemi a finestre mostrano il menu contestuale alla pressione del tasto destro (evento `mousePressed`), mentre altri lo mostrano al momento del rilascio (evento `mouseReleased`), è bene ascoltare entrambi gli eventi, controllando la condizione `isPopupTrigger()` sull'evento `MouseEvent`: esso restituisce `true` solamente se l'evento corrente è quello che provoca il richiamo del menu contestuale nella piattaforma ospite.

```
class PopupListener extends MouseAdapter {
   // tasto destro premuto (stile Motif)
   public void mousePressed(MouseEvent e) {
      if (e.isPopupTrigger()) {
            popup.show(e.getComponent(), e.getX(), e.getY());
      }
```

Figura 10.20 – *Esempio di* `JPopupMenu`

```
   }
   // tasto destro premuto e rilasciato (stile Windows)
   public void mouseReleased(MouseEvent e) {
      if (e.isPopupTrigger()) {
         popup.show(e.getComponent(), e.getX(), e.getY());
      }
   }
}
```

Questo accorgimento permette di creare programmi che rispecchiano il comportamento della piattaforma ospite, senza ambiguità che potrebbero disorientare l'utente.

Il seguente esempio crea un `JTextField` al quale aggiunge un `MouseListener` che si occupa di visualizzare un `JPopupMenu` alla pressione del tasto destro.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JPopupMenuExample extends JFrame {
   private JPopupMenu popup;

   public JPopupMenuExample() {
      super("JPopupMenuExample");
      setBounds(10, 35, 350, 120);

      JTextField textField
      = new JTextField("Premi il tasto sinistro per vedere un JPopupMenu");
      textField.setEditable(false);
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(textField);

      popup = new JPopupMenu();
         JMenuItem popupItem1 = new JMenuItem("PopupItem 1");
         JMenuItem popupItem2 = new JMenuItem("PopupItem 2");
         JMenuItem popupItem3 = new JMenuItem("PopupItem 3");
         popup.add(popupItem1);
         popup.add(popupItem2);
         popup.add(popupItem3);

      // Aggiunge un MouseListener al componente
      // che deve mostrare il menu
      MouseListener popupListener = new PopupListener();
      textField.addMouseListener(popupListener);
      setVisible(true);
   }
   class PopupListener extends MouseAdapter {
```

```
      public void mousePressed(MouseEvent e) {
         if (e.isPopupTrigger()) {
            popup.show(e.getComponent(), e.getX(), e.getY());
         }
      }
      public void mouseReleased(MouseEvent e) {
         if (e.isPopupTrigger()) {
            popup.show(e.getComponent(), e.getX(), e.getY());
         }
      }
   }
   public static void main(String[] args) {
      JPopupMenuExample window = new JPopupMenuExample();
   }
}
```

Gestione degli eventi

La gestione degli eventi nei menu è del tutto simile a quella dei pulsanti: ogni volta che si seleziona un `JMenuItem`, esso lancia un `ActionEvent` ai suoi ascoltatori. Normalmente si usa `ActionListener` per i `JMenuItem`, `ItemListener` per i `JCheckboxMenuItem`, mentre per `JRadioButtonMenuItem` è possibile usare sia l'uno che l'altro.

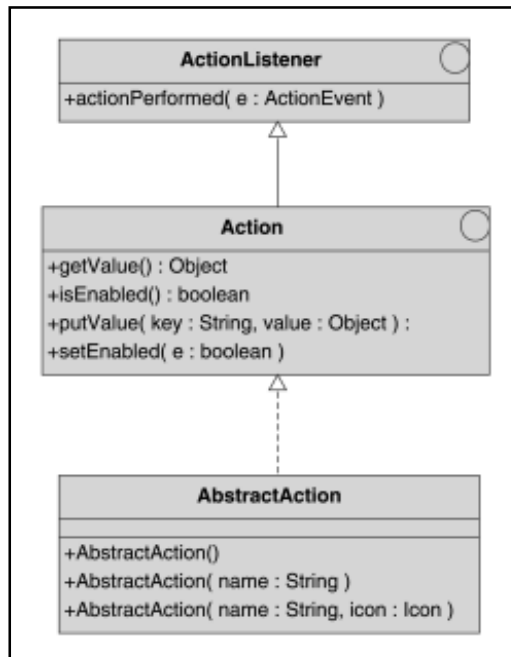
Le Action: un modo alternativo di gestire gli eventi

La maggior parte dei programmi grafici permette di accedere a una funzionalità in diverse maniere. I Word Processor, ad esempio, permettono di effettuare un *cut* su clipboard in almeno tre modi distinti: dal menu *Edit*, tramite il pulsante identificato dall'icona della forbice o tramite una voce del menu contestuale. Questa ridondanza è gradita all'utente, che ha la possibilità di utilizzare il programma secondo le proprie abitudini e il proprio grado di esperienza, ma può rivelarsi complicata da implementare per il programmatore.

Su Swing è possibile risolvere questo genere di problemi ricorrendo alle `Action`, oggetti che permettono di associare un particolare evento a un gruppo di controlli grafici, fornendo nel contempo la possibilità di gestire in modo centralizzato gli attributi e lo stato.

Descrizione dell'API

L'interfaccia `Action`, sottoclasse di `ActionListener`, eredita il metodo `actionPerformed(ActionEvent e)` con il quale si implementa la normale gestione degli eventi.

Figura 10.21 – *Gerarchia di* `Action`

Il metodo `setEnabled(boolean b)` permette di abilitare una `Action`; la chiamata a questo metodo provoca automaticamente l'aggiornamento dello stato di tutti i controlli grafici ad essa associati.

La coppia di metodi

```
Object getValue(String key)
void putValue(String key, Object value)
```

serve a leggere o a impostare coppie chiave–valore in cui la chiave è una stringa che descrive un attributo, e il valore è l'attributo stesso. Tra le possibili chiavi possiamo elencare

```
Action.NAME
```

La chiave che specifica il nome dell'azione che verrà riportato sul pulsante. Il valore corrispondente deve essere di tipo `String`.

```
Action.SHORT_DESCRIPTION
```

Specifica la descrizione dell'azione, usata nei ToolTip. Anche in questo caso si richiede un valore di tipo String.

```
Action.SMALL_ICON
```

La chiave che corrisponde all'icona di default associata a questa Action. Il valore deve essere un oggetto di tipo Icon.

Se ad esempio si desidera impostare l'icona relativa a una Action, occorre utilizzare l'istruzione putValue in questo modo:

```
action.putValue(Action.SMALL_ICON, new ImageIcon("img.gif"))
```

La classe AbstractAction, implementazione dell'interfaccia Action, fornisce alcuni costruttori che permettono di impostare le proprietà in modo più elegante.

```
AbstractAction(String name)
```

Definisce una Action con il nome specificato dal parametro.

```
AbstractAction(String name, Icon icon)
```

Definisce una Action con l'icona e il nome specificati dai parametri.

Esempio pratico

È possibile creare un oggetto Action estendendo la classe AbstractAction e fornendo il codice del metodo actionPerformed(ActionEvent e), in modo simile a quanto si farebbe per un oggetto di tipo ActionListener.

```
class MyAction extends AbstractAction {
   private Icon myIcon = new ImageIcon("img.gif");
   public MyAction() {
      super("My Action", myIcon);
   }
   public void actionPerformed(ActionEvent e) {
      // qui va il codice dell'ascoltatore
   }
}
```

È possibile definire una Action come classe anonima.

```
Action myAction = new AbstractAction("My Action", new ImageIcon("img.gif")) {
   public void actionPerformed(ActionEvent e) {
      // qui va il codice dell'ascoltatore
   }
});
```

Per abbinare una `Action` ai corrispondenti controlli grafici è sufficiente utilizzare il metodo `add(Action a)` presente su `JMenu`, `JToolBar` e `JPopupMenu`, come si vede nelle seguenti righe.

```
Action myAction = new MyAction();
JToolBar toolBar = new JToolBar();
JMenuBar menuBar = new JMenuBar();
JPopupMenu popup = new JPopupMenu();
// aggiunge un pulsante alla Tool Bar
toolBar.add(myAction);
// aggiunge un MenuItem alla Menu Bar
menuBar.add(myAction);
// aggiunge un MenuItem al Popup Menu
popup.add(myAction);
```

Dal momento che il metodo `add(Action)` restituisce il componente che viene creato, è possibile cambiarne l'aspetto anche dopo che è stato creato. Se si vuole aggiungere un `MenuItem` al menu, ma si desidera che esso sia rappresentato soltanto da una stringa di testo, senza icona, è possibile agire nel seguende modo:

```
JMenuItem mi = menuBar.add(myAction);
mi.setIcon(null);
```

Se durante l'esecuzione del programma si vogliono disabilitare i controlli abbinati a `MyAction`, è possibile farlo ricorrendo all'unica istruzione

```
myAction.setEnabled(false);
```

che provvederà a disabilitare tutti i controlli legati a `MyAction`.

Controlli avanzati

JTextField

I `JTextField` sono oggetti grafici che permettono di editare una singola riga di testo. Premendo il tasto Invio viene generato un `ActionEvent`, per segnalare agli ascoltatori

Figura 10.22 – *Un semplice campo di testo*

che il testo è stato immesso. È possibile creare un `JTextField` usando i seguenti costruttori

```
JTextField()
```

Crea un Text Field vuoto.

```
JTextField(int columns)
```

Crea un Text Field vuoto con il numero di colonne specificato dal parametro

```
JTextField(String text)
```

Crea un Text Field con il testo specificato dal parametro

```
JTextField(String text, int columns)
```

Crea un `JTextField` con il testo e il numero di colonne specificati dai parametri.

I seguenti metodi permettono di impostare o di leggere le principali proprietà dell'oggetto.

```
setText(String text)
```

Permette di settare il contenuto del campo di testo.

```
String getText()
```

Permette di leggere il contenuto del `JTextField`.

```
setColumns(int columns)
```

Imposta il numero di colonne.

```
setFont(Font f)
```

Imposta il font.

```
setHorizontalAlignment(int alignment)
```

Regola l'allineamento del testo; il parametro può assumere uno dei seguenti valori: `JTextField.LEFT`, `JTextField.CENTER`, `JTextField.RIGHT`.

L'ascoltatore di default per `JTextField` è `ActionListener`, che viene invocato alla pressione del tasto `Invio`. `JTextField` ha un metodo per impostare la stringa usata come command string per l'`ActionEvent` inviato. Se non viene specificato diversamente, tale stringa assume il valore del testo presente all'interno del componente.

```
addActionListener(ActionListener l)
```

Aggiunge un `ActionListener` al `JTextField`

```
setActionCommand(String command)
```

Imposta la "command string" dell'`ActionEvent`.

Nell'esempio seguente si vedrà come creare un `JTextField` e un ascoltatore che reagisca all'inserimento di testo.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JTextFieldExample extends JFrame {
   private JTextField textField;
```

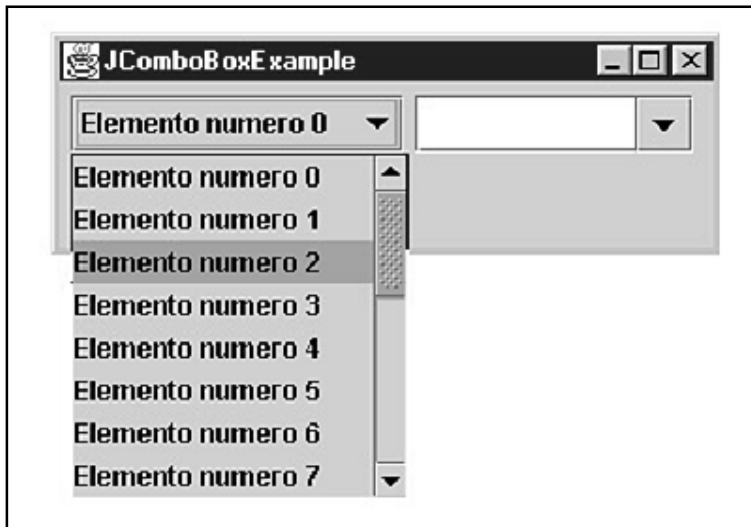
```
   private JLabel label;

   public JTextFieldExample() {
      super("JTextField");
      setSize(200, 80);
      getContentPane().setLayout(new BorderLayout());

      textField = new JTextField();
      label = new JLabel();
      getContentPane().add(BorderLayout.NORTH, textField);
      textField.addActionListener(new EnterTextListener());
      getContentPane().add(BorderLayout.SOUTH, label);

      setVisible(true);
   }
   class EnterTextListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         label.setText("Testo inserito: " + textField.getText());
         textField.setText("");
      }
   }
   public static void main(String argv[]) {
      JTextFieldExample jtf = new JTextFieldExample();
   }
}
```

JPasswordField

`JPasswordField` è una sottoclasse di `JTextField` specializzata nell'inserimento di password. Le principali differenze rispetto alla superclasse sono due: la prima è che su `JPasswordField` i caratteri digitati vengono visualizzati di default tramite asterischi (`***`); la seconda è che il testo in chiaro viene restituito sotto forma di array di char invece che come stringa. Il metodo `setEchoChar(char c)` permette di impostare qualunque carattere al posto dell'asterisco di default; il metodo `char[] getPassword()` invece restituisce il contenuto del campo di testo in chiaro, sotto forma di array di char.

JComboBox

Attraverso i `JComboBox` è possibile offrire all'utente la possibilità di effettuare una scelta a partire da una lista di elementi, anche molto lunga. A riposo, il componente si presenta come un pulsante, con l'etichetta corrispondente al valore attualmente selezionato; un click del mouse provoca la comparsa di un menu provvisto di barra laterale di scorrimento, che mette in vista le opzioni disponibili. Se si imposta un `JComboBox` come editabile, esso si comporterà a riposo come un `JTextField`, permettendo all'utente di inserire valori non presenti nella lista.

Figura 10.23 – `JComboBox` *permette di scegliere un elemento da una lista*

È possibile creare un `JComboBox` usando i seguenti costruttori; il secondo costruttore permette di inizializzare il componente con una lista di elementi di tipo `String`, `Icon` e `JLabel`.

```
JComboBox()
```

Crea un `JComboBox` vuoto.

```
JComboBox(Object[] items)
```

Crea un `JComboBox` contenente gli elementi specificati dal parametro `items`.

Alcuni metodi permettono di manipolare gli elementi dalla lista.

```
void addItem(Object anObject)
```

Aggiunge un elemento alla lista.

```
void removeItem(Object anObject)
```

Rimuove un elemento dalla lista.

```
void removeItemAt(int anIndex)
```

Rimuove l'elemento nella posizione specificata dal parametro.

```
void removeAllItems()
```

Rimuove tutti gli elementi.

```
Object getItemAt(int index)
```

Restituisce l'elemento contenuto nella posizione specificata.

```
int getItemCount()
```

Restituisce il numero di elementi contenuto nella lista.

```
void insertItemAt(Object anObject, int index)
```

Inserisce un elemento nella posizione specificata.

Per manipolare l'aspetto dell'oggetto si può ricorrere ai seguenti metodi.

```
void setMaximumRowCount(int count)
```

Imposta il numero massimo di elementi.

```
Object getSelectedItem()
```

Restituisce l'elemento attualmente selezionato.

```
void setSelectedIndex(int anIndex)
```

Seleziona l'elemento all'indice specificato.

`void setSelectedItem(Object anObject)`

Seleziona l'`Object` specificato nel parametro.

`void setEnabled(boolean b)`

Abilita/disabilita il componente.

`void setEditable(boolean aFlag)`

Imposta se il componente è editabile oppure no.

Nel seguente esempio vengono creati due `JComboBox`: uno editabile e l'altro non editabile; nel primo è possibile inserire gli elementi digitandoli direttamente dentro il componente e premendo invio. Come ascoltatori vengono usati due `ActionListener`: `EditListener` si occupa di aggiungere alla lista i nuovi elementi, mentre `SelectionListener` viene invocato da entrambi i componenti al fine di aggiornare una `JLabel` con il valore dell'elemento selezionato.

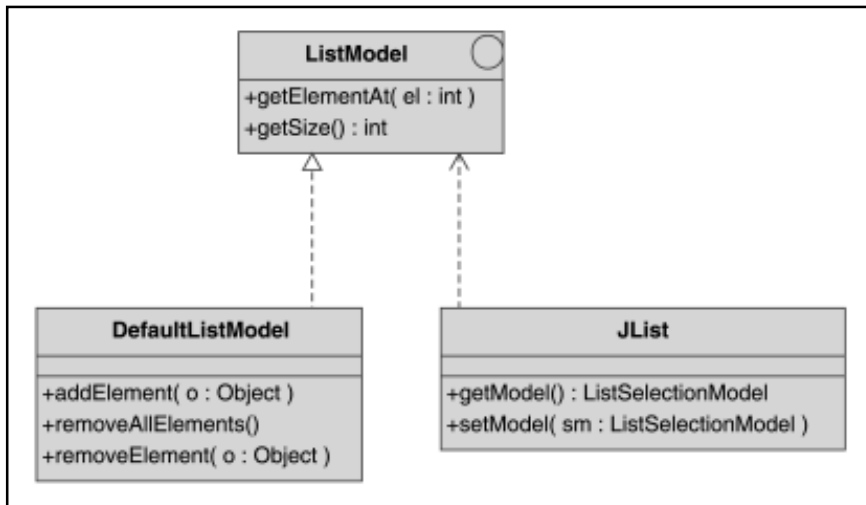
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JComboBoxExample extends JFrame {
   private JComboBox uneditableComboBox;
   private JLabel label;
   private JComboBox editableComboBox;
   private String[] items;

   public JComboBoxExample() {
      // Imposta le proprietà del Top Level Container
      super("JComboBoxExample");
      setBounds(10, 35, 300, 100);
      getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT));

      // Crea 20 elementi
      items = new String[20];
      for(int i = 0; i < 20; i++)
         items[i] = "Elemento numero " + String.valueOf(i);
```

```
      // Inizializza un ComboBox non editabile
      uneditableComboBox = new JComboBox(items);
      ActionListener selectionListener = new SelectionListener();
      uneditableComboBox.addActionListener(selectionListener);

      label = new JLabel();

      // Inizializza un JComboBox editabile
      editableComboBox = new JComboBox();
      editableComboBox.setEditable(true);
      editableComboBox.addActionListener(new EditListener());
      editableComboBox.addActionListener(selectionListener);

      getContentPane().add(uneditableComboBox);
      getContentPane().add(editableComboBox);
      getContentPane().add(label);

      setVisible(true);
   }

   class SelectionListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         JComboBox cb = (JComboBox)e.getSource();
         String selectedItem = (String)cb.getSelectedItem();
         label.setText("Selezionato: " + selectedItem);
      }
   }
   class EditListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         JComboBox cb = (JComboBox)e.getSource();
         String selectedItem = (String)cb.getSelectedItem();
         editableComboBox.addItem(selectedItem);
         editableComboBox.setSelectedItem("");
      }
   }
   public static void main(String argv[]) {
      JComboBoxExample b = new JComboBoxExample();
   }
}
```

`JList`

`JList` è un altro componente che permette di scegliere tra elementi che compongono una lista; diversamente da `JComboBox` è possibile selezionare più di un elemento per volta, utilizzando il tasto `SHIFT` per selezionare elementi contigui o il tasto `CTRL` per elementi separati.

Per utilizzare in modo completo `JList` è necessario comprendere la sua struttura interna: come si può vedere dalla fig. 10.24, `JList` mantiene gli elementi della lista in un

Figura 10.24 – *Architettura di* `JList`

oggetto conforme all'interfaccia `ListModel`. Il package `javax.swing` contiene `DefaultListModel`, un'implementazione di `JList` di uso generico, che permette di aggiungere o togliere a piacere elementi dalla lista.

Contrariamente a quanto suggerisce il nome, `DefaultListModel` *non* è il modello di default: se si crea una `JList` a partire da un vettore, esso utilizzerà un proprio `ListModel` non modificabile, al quale non potranno essere aggiunti o tolti elementi. Se si vuole creare una `JList` più flessibile, occorre procedere nel modo seguente.

```
listModel = new DefaultListModel();
listModel.addElement("Elemento 1");
listModel.addElement("Elemento 2");
...
list = new JList(listModel);
```

Per visualizzare correttamente `JList` è indispensabile montarlo all'interno di un `JScrollPane`, un pannello dotato di scrollbar, e aggiungere quest'ultimo al pannello principale; in caso contrario non sarà possibile visualizzare tutti gli elementi presenti nella lista.

```
list = new JList(listModel);
JScrollPane scroll = new JScrollPane(list);
panel.add(scroll);
```

Ecco, di seguito, i costruttori e i metodi più importanti.

`JList(ListModel dataModel)`

Crea una `JList`, che mantiene gli elementi nel modello specificato.

`JList(Object[] listData)`

Crea una `JList` a partire dal vettore di elementi che costituiscono il parametro. La lista così costituita non può essere modificata.

`void clearSelection()`

Azzera la selezione.

`int getSelectedIndex()`

Restituisce l'indice del primo elemento selezionato. Se al momento non è selezionato alcun elemento, viene restituito `-1`.

`int[] getSelectedIndices()`

Restituisce in un array gli indici degli elementi selezionati in ordine crescente.

`Object getSelectedValue()`

Restituisce il primo elemento selezionato, o `null` se al momento non è selezionato alcun elemento.

`Object[] getSelectedValues()`

Restituisce in un vettore gli elementi selezionati.

`boolean isSelectedIndex(int index)`

Permette di sapere se l'elemento all'indice specificato è selezionato o meno.

```
void setSelectedIndex(int index)
```

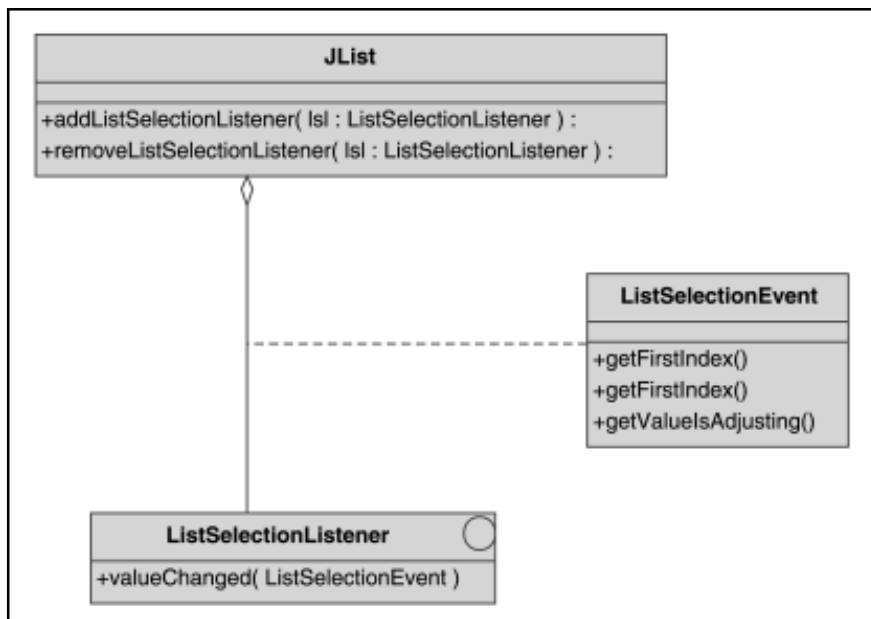
Seleziona l'elemento all'indice specificato.

```
void setSelectedIndices(int[] indices)
```

Seleziona un insieme di celle.

```
void setSelectionInterval(int anchor, int lead)
```

Seleziona un intervallo di celle.

Figura 10.25 – *Gestione degli eventi di* `JList`

```
void setSelectionMode(int selectionMode)
```

Imposta la modalità di selezione. Il parametro può assumere i valori: `JList.SINGLE_SELECTION` se si desidera che si possa selezionare un solo elemento per volta; `JList.SINGLE_INTERVAL_SELECTION` se si vuole permettere la selezione di un singolo intervallo per volta; `JList.MULTIPLE_INTERVAL_SELECTION` se non si vuole porre restrizioni al numero di elementi o intervalli selezionabili.

Ogni volta che l'utente seleziona un elemento, viene notificato un `ListSelectionEvent` ai `ListSelectionListener` registrati. I metodi di `ListSelectionEvent` permettono di conoscere gli indici di inizio e di fine della selezione. Attraverso il metodo `boolean getValueIsAdjusting()` è possibile sapere se l'utente sta ancora operando sulla selezione, o se ha terminato rilasciando il pulsante del mouse.

```
void addListSelectionListener(ListSelectionListener listener)
```

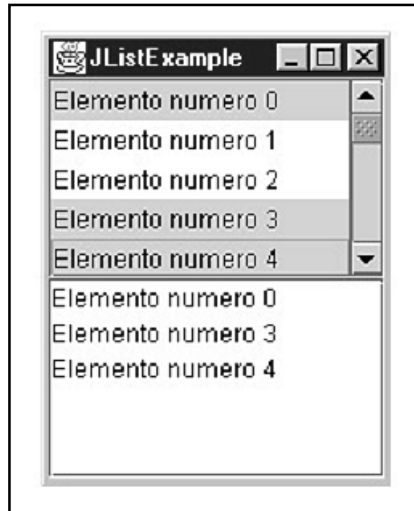
Registra un `ListSelectionListener`, che verrà notificato ogni qual volta viene selezionato un elemento (o un gruppo di elementi) dalla `JList`.

```
void removeListSelectionListener(ListSelectionListener listener)
```

Rimuove l'ascoltatore specificato.

L'esempio seguente crea un `JList` con 20 elementi, selezionabili a intervalli non contigui, e un'area di testo non modificabile che elenca gli elementi attualmente selezionati.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class JListExample extends JFrame {

   private JList list;
   private JTextArea output;

   public JListExample() {
      super("JListExample");
      setSize(170, 220);
      getContentPane().setLayout(new GridLayout(0, 1));

      // Crea 20 elementi
```

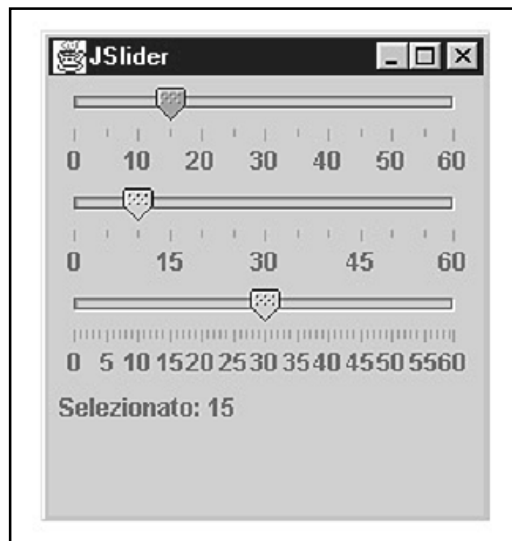
Figura 10.26 – *Tenendo premuto il tasto CTRL è possibile selezionare elementi non contigui*

```
    String[] items = new String[20];
    for(int i = 0; i < 20;i++)
       items[i] = "Elemento numero " + String.valueOf(i);

    // Inizializza una JList
    list = new JList(items);
    list. setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    ListSelectionListener selectionListener = new SelectionListener();
    list.addListSelectionListener(selectionListener);

    // Crea la TextArea di output
    output = new JTextArea();
    output.setEditable(false);

    // assembla la GUI
    getContentPane().add(new JScrollPane(list));
    getContentPane().add(new JScrollPane(output));
    setVisible(true);
 }
 class SelectionListener implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent e) {
       if(!e.getValueIsAdjusting()) {
          JList list = (JList)e.getSource();
          output.setText("");
          Object[] selectedItems = list.getSelectedValues();
          for(int i = 0; i < selectedItems.length; i++)
```

```
                output.append(selectedItems[i].toString() + "\n");
            }
        }
    }
  public static void main(String argv[]) {

    JListExample b = new JListExample();
  }
}
```

`JSlider`

`JSlider` è un cursore a slitta, che permette di inserire in maniera continua valori numerici compresi tra un massimo e un minimo, eliminando di fatto la possibilità di inserire valori scorretti. L'insieme di controlli AWT comprende un componente simile, `Scrollbar`; le caratteristiche presenti in `JSlider` sono tuttavia molto più avanzate, e offrono maggiori possibilità di personalizzazione.

Le proprietà più importanti di un `JSlider` sono il valore minimo, il valore massimo e l'orientamento, che può essere orizzontale o verticale; i costruttori permettono di specificare tutte queste proprietà al momento della creazione dell'oggetto. Altre importanti proprietà permettono di specificare il formato del righello graduato che decora lo slider. Le righe di codice seguenti chiariranno meglio l'uso di questo componente.

Figura 10.27 – *Alcuni esempi di* `JSlider`

```
slider = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(5);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
```

Il costruttore crea un `JSlider` orizzontale, la cui scala varia tra 0 e 60 con il cursore posizionato sul 15; seguono due metodi che impostano la spaziatura tra le tacche del righello: il primo imposta la spaziatura tra le tacche più grandi, il secondo tra quelle più piccole. Gli ultimi due metodi attivano il disegno del righello e della guida numerata, normalmente disattivati.

Questi pochi metodi permettono di creare un gran numero di slider, come è possibile vedere in fig. 10.27.

Esiste la possibilità di personalizzare ulteriormente l'aspetto del componente, specificando etichette non regolari. Per farlo bisogna chiamare il metodo `setLabelTable(HashTable h)`, passando come parametro una `HashTable` contenente coppie chiave–valore composte da un oggetto di tipo `Int` e un `Component`: ogni `Component` verrà disegnato in corrispondenza della tacca specificata dell'intero passato come chiave. Nelle righe seguenti viene illustrato come creare un `JSlider` con etichette testuali; ovviamente è possibile utilizzare al posto delle label qualunque tipo di componente, ad esempio `JLabel` contenenti icone, o addirittura pulsanti programmati per riposizionare il cursore su valori preimpostati.

```
slider = new JSlider(JSlider.VERTICAL, 0, 70, 15);
slider.setMajorTickSpacing(10);
slider.setPaintTicks(true);

Hashtable labelTable = new Hashtable();
labelTable.put(new Integer(0), new JLabel("Silence"));
labelTable.put(new Integer(10), new JLabel("Low"));
labelTable.put(new Integer(30), new JLabel("Normal"));
labelTable.put(new Integer(70), new JLabel("Loud!"));
slider.setLabelTable(labelTable);
```

Ogni volta che si manipola un `JSlider`, viene generato un `ChangeEvent`. Un ascoltatore di tipo `ChangeListener` deve implementare il metodo `stateChanged(ChangeEvent e)`, come si vede nell'esempio che seguirà. È possibile conoscere lo stato di un `JSlider` attraverso due metodi: `getValue()` restituisce il valore intero su cui il cursore è attualmente posizionato; `getValueIsAdjusting()` restituisce `true` se l'azione di modifica è tuttora in corso. Un ascoltatore come il seguente effettua un'azione solamente quando il cursore viene rilasciato, e scarta tutti gli eventi di aggiustamento.

```
class SliderChangeListener implements ChangeListener {
   public void stateChanged(ChangeEvent e) {
```

```
      JSlider slider = (JSlider)e.getSource();
      if(!slider.getValueIsAdjusting())
         label.setText("Selezionato: " + String.valueOf(slider.getValue()));
   }
}
```

`JSlider(int orientation, int min, int max, int value)`

Crea un `JSlider` con l'orientamento specificato dal parametro `orientation`, che può assumere i due valori `JSlider.VERTICAL` o `JSlider.HORIZONTAL`. I parametri `min` e `max` specificano i valori limite della scala, mentre `value` indica la posizione iniziale del cursore.

`void addChangeListener(ChangeListener l)`

Aggiunge un `ChangeListener` al componente.

`void removeChangeListener(ChangeListener l)`

Rimuove il `ChangeListener` dalla lista degli ascoltatori.

`void setOrientation(int orientation)`

Imposta l'orientamento dello slider; il parametro può assumere i valori `JSlider.HORIZONTAL` o `JSlider.VERTICAL`.

`void setValue(int value)`

Imposta il cursore sul valore specificato dal parametro.

`int getValue()`

Restituisce il valore corrente dello slider.

`void setInverted(boolean b)`

Se il parametro è `true`, la scala viene disegnata da destra a sinistra (o dal basso all'alto) invece che il contrario.

`void setMinimum(int min)`

Imposta il valore minimo della scala.

`void setMaximum(int max)`

Imposta il valore massimo della scala.

`void setPaintTicks(boolean b)`

Attiva o disattiva la visualizzazione del righello.

`void setMajorTickSpacing(int)`

Imposta la spaziatura tra le tacche grandi del righello.

`void setMinorTickSpacing(int)`

Imposta la spaziatura tra le tacche piccole del righello.

`void setPaintLabels(boolean b)`

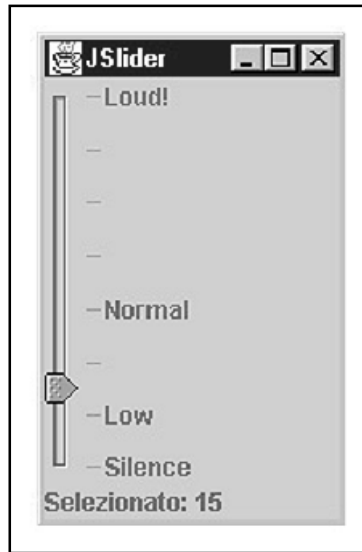
Attiva o disattiva la visualizzazione delle etichette.

`void setLabelTable(HashTable Dictionary)`

Imposta una tabella hash contenente informazioni per disegnare la tabella delle etichette.

`Hashtable createStandardLabels(int increment)`

Crea una tabella hash che disegna etichette con l'incremento specificato dal parametro.

Figura 10.28 – *Con il metodo* `setLabelTable()` *è possibile personalizzare il righello*

`JTextArea`

Il package Swing offre un set completo di componenti di testo: `JTextComponent`, `JTextField`, `JPasswordField`, `JTextArea`, `JEditorPane`, `JTextPane`.

Il primo componente dell'elenco è la superclasse astratta da cui discendono tutti gli altri Text Component: esso definisce l'interfaccia di programmazione e una prima implementazione di tutte le funzionalità normalmente presenti su editor di testo, come ad esempio la possibilità di inserire caratteri da tastiera; quella di spostare il cursore in qualunque punto del documento utilizzando il mouse o i tasti cursore; la possibilità di operare selezioni su gruppi contigui di caratteri al fine di eseguire su di essi una qualunque operazione di editing; i metodi per operare sulla clipboard di sistema (che permette di copiare e incollare testi anche tra programmi Java e programmi nativi) e quelli per leggere il documento da uno stream di input o per inviarlo a uno di output.

Nei paragrafi precedenti è stato analizzato il funzionamento di `JTextField`, un componente che permette di editare una singola riga di testo; si prenderà ora in considerazione `JTextArea`, un oggetto grafico da utilizzare quando si intende lavorare su testi di lunghezza media, con l'unica limitazione di permettere l'utilizzo di un unico carattere, un unico stile ed un unico colore per tutto il documento.

Nei casi in cui si desideri lavorare su documenti complessi, senza subire limiti nella scelta di font, stile o dimensioni, è possibile utilizzare `JEditorPane` e `JTextPane`, oggetti grafici che permettono di lavorare su formati testo come RTF o HTML e con la

possibilità aggiuntiva di arricchire i testi con immagini o componenti attivi. La complessità di questa coppia di componenti impedisce una trattazione in questa sede (essi richiederebbero un libro a sé stante); uno sguardo approfondito a `JTextArea` sarà comunque più che sufficiente per rompere il ghiaccio con questa famiglia di componenti.

`JTextArea` API

La via più breve per creare una `JTextArea` è attraverso il costruttore privo di parametri; altri costruttori permettono di specificare il numero di righe e di colonne e la composizione iniziale del testo. È importante ricordare che `JTextArea`, al contrario del suo equivalente AWT, non dispone di barre laterali di scorrimento; per questa ragione è indispensabile, all'atto della sua creazione, inserirlo dentro un `JScrollPane`, come si può vedere nelle righe seguenti.

```
...
JTextArea ta = new JTextArea();
JScrollPane scroll = new JScrollPane(ta);
getContentPane().add(BorderLayout.CENTER, scroll);
...
```

```
JTextArea()
```

Crea una `JTextArea`.

```
JTextArea(String text, int rows, int columns)
```

Crea una `JTextArea` con il testo specificato dalla stringa `text`, e le dimensioni specificate dai parametri rows e columns.

È possibile editare il testo nel componente direttamente con la tastiera e il mouse, o da programma ricorrendo ad alcuni metodi. Questi metodi richiedono come parametro un valore intero, che specifica la posizione del cursore rispetto all'inizio del documento: se la text area in questione contenesse nella prima riga la frase "La vispa Teresa" e nella seconda "avea tra l'erbetta", sarebbe possibile dire che la parola "Teresa" è compresa tra gli offset `9` e `15`, mentre "erbetta" si trova tra gli indici `27` e `34` (va contato anche il ritorno carrello).

```
void setText(String t)
```

Cancella il precedente contenuto del componente e lo rimpiazza con la stringa specificata.

```
String getText()
```

Restituisce il testo contenuto all'interno del componente.

```
String getText(int offs, int len)
```

Restituisce il testo che parte dalla posizione specificata dal primo parametro, della lunghezza specificata dal secondo.

```
void insert(String str, int pos)
```

Inserisce il testo contenuto nella stringa nella posizione specificata dal parametro.

```
void append(String str)
```

Inserisce il contenuto della stringa in coda al documento.

```
void replaceRange(String str, int start, int end)
```

Rimpiazza la sezione di testo compresa tra `start` ed `end` con la stringa specificata. In alternativa è possibile inizializzare il componente a partire da uno stream.

```
void read(Reader in, Object desc)
```

Legge il testo da uno stream (il secondo parametro può essere messo a `null`).

```
void write(Writer out)
```

Salva il testo su uno stream.

Un gruppo di metodi permette di conoscere la posizione attuale del cursore ed eventualmente modificarla: tra questi metodi ci sono anche quelli che permettono di operare selezioni, ovvero di mettere in evidenza una porzione di testo contigua, indicandone l'inizio e la fine.

```
int getCaretPosition()
```

Restituisce la posizione del cursore a partire dall'inizio del documento.

```
void setCaretPosition(int position)
```

Sposta il cursore alla posizione specificata.

```
void moveCaretPosition(int pos)
```

Muove il cursore fino alla posizione specificata, evidenziando il testo tra la nuova posizione e quella precedente.

```
void select(int selectionStart, int selectionEnd)
```

Seleziona il testo situato tra la posizione iniziale e quella finale, specificate dai parametri.

```
String getSelectedText()
```

Restituisce il testo contenuto nella selezione.

```
void replaceSelection(String content)
```

Rimpiazza l'attuale selezione con il testo contenuto nella stringa.

```
int getSelectionStart()
```

Restituisce il punto di inizio della selezione.

```
int getSelectionEnd()
```

Restituisce il punto finale della selezione.

Si può operare sulla clipboard di sistema attraverso i metodi:

```
void copy()
void cut()
void paste()
void selectAll()
```

Infine è possibile impostare alcune importanti proprietà grazie ai metodi riportati di seguito.

```
void setFont(Font font)
```

Imposta il font.

```
void setForeground(Color fg)
```

Imposta il colore del testo.

```
void setBackground(Color bg)
```

Imposta il colore dello sfondo.

```
void setEditable(boolean b)
```

Imposta il componente come editabile o non editabile.

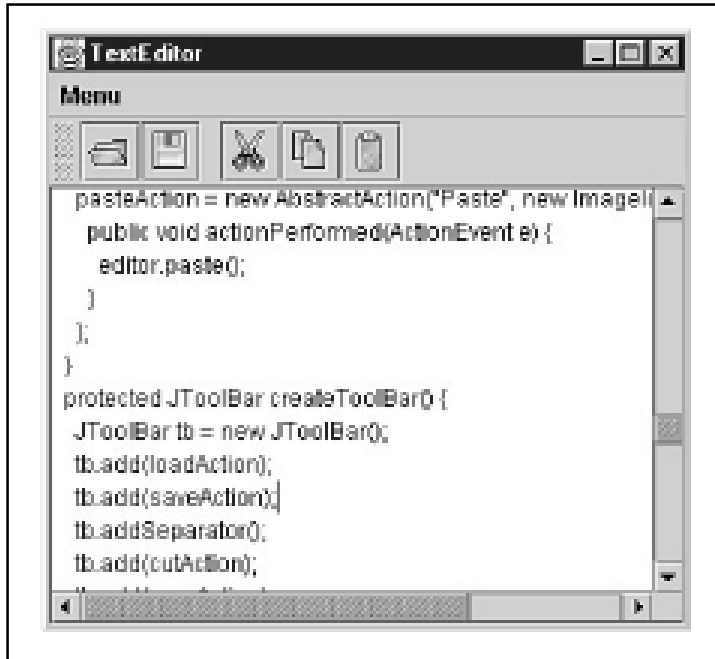
Sviluppare un'applicazione grafica complessa

Le nozioni apprese fino ad ora permettono di affrontare lo studio di un'applicazione grafica di una discreta complessità. Le seguenti righe permettono di realizzare un piccolo editor di testo perfettamente funzionante, utilizzando `JTextArea`, una `JToolBar`, una `JMenuBar`, un `JFileChooser` e mostrando un utilizzo pratico delle `Action`. Viene inoltre illustrato, all'interno dei metodi `loadText()` e `saveText()`, come sia possibile inizializzare un `Text Component` a partire da un file su disco.

```
import javax.swing.*;
import javax.swing.text.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class TextEditor extends JFrame {
   private JTextComponent editor;
   private JFileChooser fileChooser;
   protected Action loadAction;
   protected Action saveAction;
   protected Action cutAction;
   protected Action copyAction;
   protected Action pasteAction;
   public TextEditor() {
      super("TextEditor");
      setSize(300, 300);
      createActions();
      JMenuBar menuBar = createMenuBar();
      JToolBar toolBar = createToolBar();
      editor = createEditor();
      JComponent centerPanel = createCenterComponent();
      getContentPane().add(BorderLayout.NORTH, toolBar);
      getContentPane().add(BorderLayout.CENTER, centerPanel);
      setJMenuBar(menuBar);
      fileChooser = new JFileChooser();
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
   protected void createActions() {
      loadAction = new AbstractAction("Open", new ImageIcon("Open24.gif")) {
         public void actionPerformed(ActionEvent e) {
            loadText();
         }
      };
      saveAction = new AbstractAction("Save", new ImageIcon("Save24.gif")) {
         public void actionPerformed(ActionEvent e) {
            saveText();
         }
      };
      cutAction = new AbstractAction("Cut", new ImageIcon("Cut24.gif")) {
         public void actionPerformed(ActionEvent e) {
            editor.cut();
         }
      };
      copyAction = new AbstractAction("Copy", new ImageIcon("Copy24.gif")) {
         public void actionPerformed(ActionEvent e) {
            editor.copy();
         }
      };
      pasteAction = new AbstractAction("Paste", new ImageIcon("Paste24.gif")) {
```

```
            public void actionPerformed(ActionEvent e) {
               editor.paste();
            }
         };
      }
      protected JToolBar createToolBar() {
         JToolBar tb = new JToolBar();
         tb.add(loadAction);
         tb.add(saveAction);
         tb.addSeparator();
         tb.add(cutAction);
         tb.add(copyAction);
         tb.add(pasteAction);
         return tb;
      }
      protected JMenuBar createMenuBar() {
         JMenu menu = new JMenu("Menu");
            menu.add(loadAction);
            menu.add(saveAction);
            menu.addSeparator();
            menu.add(cutAction);
            menu.add(copyAction);
            menu.add(pasteAction);
         JMenuBar menuBar = new JMenuBar();
         menuBar.add(menu);
         return menuBar;
      }
      protected JComponent createCenterComponent() {
         if(editor == null)
            editor = createEditor();
         return new JScrollPane(editor);
      }
      protected JTextComponent createEditor() {
         return new JTextArea();
      }
      public void loadText() {
         int response = fileChooser.showOpenDialog(this);
         if(response == JFileChooser.APPROVE_OPTION) {
            try {
               File f = fileChooser.getSelectedFile();
               Reader in = new FileReader(f);
               editor.read(in, null);
               setTitle(f.getName());
            }
            catch(Exception e) {}
         }
      }
      public void saveText() {
         int response = fileChooser.showSaveDialog(this);
```

Figura 10.29 – *Con appena un centinaio di righe è possibile realizzare un utile text editor*

```
      if(response == JFileChooser.APPROVE_OPTION) {
         try {
            File f = fileChooser.getSelectedFile();
            Writer out = new FileWriter(f);
            editor.write(out);
            setTitle(f.getName());
         }
         catch(Exception e) {}
      }
   }
   public static void main(String argv[]) {
      TextEditor t = new TextEditor();
   }
}
```

Il metodo `createActions()` riesce a definire cinque classi in appena trenta righe di codice facendo uso delle classi anonime. L'uso di classi anonime in questo contesto è giustificato dal desiderio di rendere il programma molto compatto; su progetti di dimensioni maggiori si consiglia comunque di ricorrere al più chiaro costrutto delle classi interne.

Questo programma vuole anche dare una dimostrazione di costruzione modulare di interfacce grafiche: come si può notare, il costruttore genera gli elementi dell'interfaccia grafica ricorrendo ad un gruppo di metodi fabbrica (*Factory Methods*), vale a dire metodi `protected` caratterizzati dal prefisso `create` — `createToolBar()`, `createMenuBar()`, `createCenterComponent()` e `createEditor()` — i quali restituiscono il componente specificato dal loro nome. Questa scelta offre la possibilità di creare sottoclassi del programma che implementino una differente composizione della GUI semplicemente sovrascrivendo questi metodi, e lasciando inalterato il costruttore. Se ad esempio si disponesse di un'ipotetica `HTMLTextArea` e si volesse creare un editor HTML, basterebbero appena cinque righe per adattare il nostro programma a una nuova funzione.

```
public class HTMLTextEditor extends TextEditor {
   protected JTextComponent createEditor() {
      return new HTMLTextArea();
   }
}
```

In uno dei prossimi paragrafi verrà presentato un ulteriore esempio, che permetterà di apprezzare i vantaggi di questa scelta di progettazione.

Pannelli specializzati

Si è visto come sia possibile creare interfacce grafiche innestando pannelli uno dentro l'altro. La scelta di pannelli disponibili su Swing non è limitata al semplice `JPanel`, ma include pannelli specializzati nel trattamento di casi particolari.

JSplitPane

`JSplitPane` è un pannello formato da due aree, separate da una barra mobile; al suo interno è possibile disporre una coppia di componenti, affiancati lateralmente o uno sopra l'altro. Il divisore può essere trascinato per impostare l'area da assegnare a ciascuno dei due componenti, rispettando la dimensione minima dei componenti in esso contenuti. Usando `JSplitPane` in abbinamento a `JScrollPane` si può ottenere una coppia di pannelli ridimensionabili.

Il seguente programma crea una finestra con uno `JSplitPane` al suo interno: nel pannello superiore monta un'immagine JPEG, in quello inferiore una Text Area sulla quale si possono annotare commenti. Per avviarlo è necessario specificare sulla riga di comando il percorso di un file JPEG o GIF (ad esempio, `java JSplitDemo c:\immagine.jpg`).

```
import javax.swing.*;
```

```
import java.awt.*;

public class JSplitDemo extends JFrame {

   public JSplitDemo(String fileName) {
      super("JSplitPane");
      setSize(300, 250);

      // costruisce un pannello contenente un'immagine
      ImageIcon img = new ImageIcon(fileName);
      JLabel picture = new JLabel(img);
      JScrollPane pictureScrollPane = new JScrollPane(picture);

      // crea un pannello contenente una TextArea
      JTextArea comment = new JTextArea();
      JScrollPane commentScrollPane = new JScrollPane(comment);

      // Crea uno SplitPane verticale con i due pannelli al suo interno
      JSplitPane splitPane  = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                                             pictureScrollPane, commentScrollPane);
      splitPane.setOneTouchExpandable(true);
      splitPane.setDividerLocation(190);
      splitPane.setContinuousLayout(true);
```

Figura 10.30 – *Un esempio di Split Pane*

```
      // aggiunge lo SplitPane al Frame principale
      getContentPane().add(splitPane);
      setVisible(true);
   }
   public static void main(String argv[]) {

      if(argv.length == 1) {
         JSplitDemo b = new JSplitDemo(argv[0]);
      }
      else
         System.out.println("usage JSplitDemo <filename>");
   }
}
```

`JSplitPane` API

Nell'esempio si è fatto ricorso a un costruttore che permette di impostare le più importanti proprietà dello `SplitPane` con un'unica istruzione. Ovviamente sono disponibili costruttori con un numero inferiore di parametri.

```
public JSplitPane(int orientamento, Component leftComponent,
                  Component rightComponent)
```

Crea un `JSplitPane` con l'orientamento specificato dal primo parametro e i componenti specificati dal secondo e dal terzo. Il parametro orientamento può assumere i valori `JSplitPane.HORIZONTAL_SPLIT` o `JSplitPane.VERTICAL_SPLIT`.

Per specificare l'orientamento o per posizionare la coppia di componenti all'interno dello SplitPane si può anche ricorrere ai metodi che seguono.

```
void setOrientation(int orientation)
```

Imposta l'orientamento del divisore.

```
void setBottomComponent(Component comp)
```

Imposta il componente inferiore.

```
void setTopComponent(Component comp)
```

Imposta il componente superiore.

`void setRightComponent(Component comp)`

Imposta il componente di destra.

`void setLeftComponent(Component comp)`

Imposta il componente di sinistra.

Un gruppo di tre metodi permette di specificare la posizione del divisore.

`void setDividerLocation(int location)`

Imposta la posizione assoluta del divisore.

`void setDividerLocation(double proportionalLocation)`

Imposta il divisore dividendo in modo proporzionale lo spazio disponibile tra i due componenti. Se il valore è `0.5` il divisore verrà posto a metà (questo metodo funziona correttamente solo se il pannello è visibile).

`void setResizeWeight(double)`

Specifica come distribuire lo spazio che si viene a creare quando il componente viene ridimensionato; se si imposta un valore di `0.5` lo spazio in più viene diviso in maniera uguale tra i due componenti.

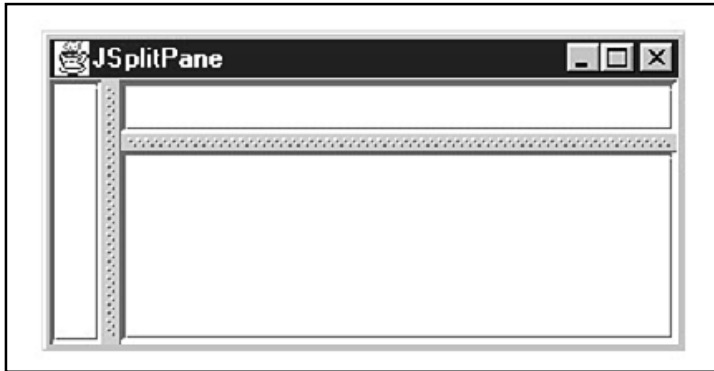
Per modificare l'aspetto fisico del pannello sono disponibili le seguenti possibilità:

`void setDividerSize(int)`

Imposta la dimensione in pixel della barra di divisione.

`void setOneTouchExpandable(boolean)`

Attivando questa proprietà, sulla barra apparirà una coppia di pulsanti a freccia che permettono di espandere o collassare il divisore con un semplice click.

Figura 10.31 – *È possibile creare Split Pane multipli inserendone uno dentro l'altro*

```
void setContinuousLayout(boolean)
```

Specifica se si desidera che il pannello venga ridisegnato durante il posizionamento del divisore.

È possibile inserire i `JSplitPane` uno dentro l'altro, nel caso si desideri avere più di due aree ridimensionabili. Ad esempio le seguenti istruzioni creano uno Split Pane orizzontale contenente una Text Area e uno Split Pane verticale, il quale contiene a sua volta due Text Area.

```
JScrollPane scroll1 = new JScrollPane(new JTextArea());
JScrollPane scroll2 = new JScrollPane(new JTextArea());
JScrollPane scroll3 = new JScrollPane(new JTextArea());

JSplitPane internalSplit
= new JSplitPane(JSplitPane.VERTICAL_SPLIT, scroll1, scroll2);
JSplitPane externalSplit
= new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, scroll3, internalSplit );
```

JTabbedPane

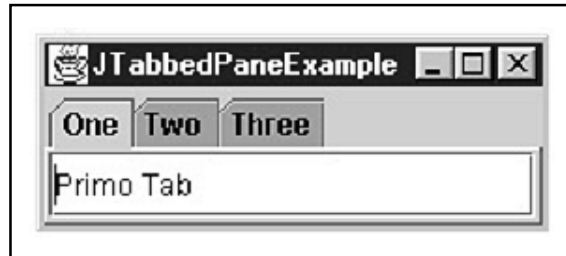
`JTabbedPane` permette a diversi componenti di condividere lo stesso spazio sullo schermo; l'utente può scegliere su quale componente operare premendo il Tab corrispondente. Il tipico uso di questo componente è nei pannelli di controllo, nei quali si assegna ad ogni Tab la gestione di un insieme di funzioni differente.

Oltre all'innegabile utilità, questo componente presenta una modalità di impiego straordinariamente semplice: è sufficiente creare il pannello ed aggiungervi i vari componenti

usando il metodo `addTab(String title, Component c)`, in cui il primo parametro specifica l'etichetta del Tab, e il secondo passa il componente. Il passaggio da un Tab all'altro viene ottenuto cliccando con il mouse sul Tab desiderato, senza bisogno di gestire gli eventi in modo esplicito.

Nell'esempio seguente viene creato un `JTabbedPane` al quale vengono aggiunti tre Tab, ognuno dei quali contiene un componente grafico diverso. L'esempio mostra anche un esempio di gestione degli eventi: al cambio di Tab viene aggiornato il titolo della finestra.

```
import javax.swing.*;
import javax.swing.event.*;

public class JTabbedPaneExample extends JFrame {
   private JTabbedPane tabbedPane;
   public JTabbedPaneExample() {
      super("JTabbedPaneExample");
      tabbedPane = new JTabbedPane();

      JTextField tf = new JTextField("primo Tab");
      JButton b = new JButton("secondo Tab");
      JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
      tabbedPane.addChangeListener(new TabListener());
      tabbedPane.addTab("uno", tf );
      tabbedPane.addTab("due", b );
      tabbedPane.addTab("tre", slider);

      getContentPane().add(tabbedPane);
      pack();
      setVisible(true);
   }
   public class TabListener implements ChangeListener {
      public void stateChanged(ChangeEvent e) {
         int pos = tabbedPane.getSelectedIndex();
         String title = tabbedPane.getTitleAt(pos);
```

Figura 10.32 – *Un semplice JTabbedPane*

```
            setTitle(title);
        }
    }
    public static void main(String[] args) {
        JTabbedPaneExample te = new JTabbedPaneExample();
    }
}
```

Nell'esempio è stato creato un `JTabbedPane` e sono stati inseriti al suo interno tre Tab, ognuno dei quali contiene un componente. Ovviamente è possibile inserire all'interno di un Tab un intero pannello con tutto il suo contenuto:

```
...
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(BorderLayout.NORTH, new Button("nord"));
...
panel.add(BorderLayout.SOUTH, new Button("sud"));
tabbedPane.addTab("Pannello", panel);
...
```

`JTabbedPane` API

Per creare un `JTabbedPane` è possibile ricorrere ai costruttori di seguito riportati.

```
JTabbedPane()
```

Crea un `JTabbedPane`.

```
JTabbedPane(int tabPlacement)
```

Crea un `JTabbedPane` con i Tab posizionati in alto, in basso, a destra o a sinistra secondo il valore del parametro: `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT`, o `JTabbedPane.RIGHT`.

Per aggiungere o togliere componenti è disponibile un gruppo di metodi.

```
void addTab(String title, Component component)
```

Aggiunge un componente su un Tab con il titolo specificato.

`void addTab(String title, Icon icon, Component component, String tip)`

Aggiunge un componente con il titolo, l'icona e il ToolTip specificati dai parametri.

`void remove(Component component)`

Rimuove il Tab contenete il componente specificato.

`void removeAll()`

Rimuove tutti i Tab.

Un gruppo di metodi permette di manipolare le proprietà dei Tab.

`Component getSelectedComponent()`

Restituisce il componente attualmente selezionato.

`void setSelectedComponent(Component c)`

Imposta il Tab che contiene il componente specificato.

`int getSelectedIndex()`

Restituisce l'indice del componente attualmente selezionato.

`void setSelectedIndex(int index)`

Imposta il tab che si trova nella posizione specificata dal parametro.

`int getTabCount()`

Restituisce il numero di Tab presenti nel Tabbed Pane.

`int indexOfComponent(Component component)`

Restituisce l'indice del componente passato come parametro.

```
String getTitleAt(int index)
```

Restituisce l'etichetta del Tab nella posizione specificata.

La gestione degli eventi su `JTabbedPane` è abbastanza limitata, dal momento che gli oggetti di tipo `ChangeEvent` non contengono nessuna informazione sul tipo di evento che li ha generati (in pratica è impossibile, per un ascoltatore, distinguere tra eventi di selezione, di aggiunta o di rimozione di tab).

È possibile aggiungere o rimuovere un ascoltatore usando i metodi:

```
void addChangeListener(ChangeListener l)
```

```
void removeChangeListener(ChangeListener l)
```

Gli ascoltatori di tipo `ChangeListener` richiedono l'implementazione del metodo `void stateChanged(ChangeEvent e)`. Gli eventi di tipo `ChangeEvent` contengono il solo metodo `Object getSource()`, che permette di ottenere un riferimento all'oggetto che ha generato l'evento; per conoscere i dettagli dell'evento (numero di Tab, titolo ecc.) è necessario interrogare direttamente il componente sorgente.

`JDesktopPane` e `JInternalFrame`

L'uso combinato di `JDesktopPane` e `JInternalFrame` apre, al programmatore Java, un universo di possibilità: attraverso questi componenti è possibile realizzare ambienti tipo desktop, dotati di finestre interne, icone e oggetti grafici.

`JInternalFrame` è un oggetto grafico molto simile, nell'aspetto, nel comportamento e nell'interfaccia di programmazione, a un `JFrame`. La differenza più consistente tra questi due oggetti è che `JInternalFrame`, non essendo un Top Level Container, non può comportarsi come radice di una gerarchia di contenimento: per visualizzare oggetti di questo tipo è necessario creare un'istanza `JDesktopPane` e aggiungervi oggetti di tipo `JInternalFrame` usando il metodo `add(component c)`.

```
JInternalFrame frame = new JInternalFrame("Frame");
desktop = new JDesktopPane();

frame.setSize(120, 80);
desktop.add(frame);
frame.setVisible(true);
```

Le finestre create all'interno di `JDesktopPane` si comportano come finestre di sistema, con l'unica differenza di essere confinate all'interno di questo pannello: l'utente può spostarle, allargarle, ridurle a icona e chiuderle, ma non può in nessun caso trascinarle al

di fuori dell'area a loro assegnata. Si noti che è obbligatorio assegnare una dimensione al `JInternalFrame`, e che è necessario chiamare il metodo `setVisible(true)` se si vuole rendere visibile la nuova finestra. È consigliabile anche prevedere una politica di posizionamento altrimenti tutte le finestre interne verranno posizionate nel margine in alto a sinistra del `JDesktopPane`.

`JDesktopPane` API

Con pochi metodi è possibile creare oggetti di tipo `JDesktopPane` e impostarne le proprietà.

`JDesktopPane()`

Crea un `JDesktopPane`.

`JInternalFrame[] getAllFrames()`

Restituisce tutti i `JInternalFrames` contenuti.

`JInternalFrame getSelectedFrame()`

Restituisce il `JInternalFrame` attivo, o `null` se nessun `JInternalFrame` è attivo al momento.

`void setDragMode(int dragMode)`

Imposta lo stile di trascinamento per il Desktop Pane. Sono disponibili due scelte: `JDesktopPane.LIVE_DRAG_MODE` fa sì che il contenuto del frame rimanga visibile durante il trascinamento; `JDesktopPane.OUTLINE_DRAG_MODE` invece fa sì che durante il trascinamento sia spostata soltanto la sagoma del frame, permettendo un refresh più rapido.

`void setSelectedFrame(JInternalFrame f)`

Imposta il `JInternalFrame` attivo.

`JInternalFrame` API

L'interfaccia di programmazione di `JInternalFrame` è abbastanza complessa, e ri-

calca il prototipo dell'API `JFrame`. Rispetto a quest'ultima, l'API `JInternalFrame` fornisce un maggior controllo sulle proprietà della barra del titolo e sui controlli che essa deve contenere per chiudere, espandere, ridurre a icona e ridimensionare la finestra; il seguente costruttore permette di impostare in un'unica istruzione tutte queste proprietà:

```
JInternalFrame(String title, boolean resizable, boolean closable,
               boolean maximizable, boolean iconifiable)
```

Crea un `JInternalFrame` con le proprietà specificate dai parametri.

I seguenti metodi sono equivalenti a quelli presenti su `JFrame`.

```
Container getContentPane()
```

Restituisce il Content Pane di questo `JInternalFrame`.

```
void setContentPane(Container c)
```

Imposta il content pane.

```
void pack()
```

Fa in modo che i componenti interni vengano impostati alla dimensione ideale.

```
void setDefaultCloseOperation(int operation)
```

Imposta l'azione da eseguire alla pressione del tasto `close`. Sono disponibili le seguenti scelte: `WindowConstants.DO_NOTHING_ON_CLOSE` non ottiene alcun effetto; `WindowConstants.HIDE_ON_CLOSE` nasconde la finestra ed è l'impostazione di default; `WindowConstants.DISPOSE_ON_CLOSE` distrugge la finestra; `void setJMenuBar(JMenuBar m)` imposta la `JMenuBar`; `void setTitle(String title)` imposta il titolo del `JInternalFrame`.

I metodi presenti invece esclusivamente su `JInternalFrame` sono qui elencati.

```
void setSelected(boolean selected)
```

Seleziona o deseleziona il `JInternalFrame`.

`JDesktopPane getDesktopPane()`

Restituisce il `JDesktopPane` che contiene questo `JInternalFrame`.

`Component getFocusOwner()`

Restituisce il `JInternalFrame` che ha il fuoco.

`void setFrameIcon(Icon icon)`

Imposta l'icona della Title Bar.

`void toBack()`

Spinge "sotto" questo `JInternalFrame`.

`void toFront()`

"Porta davanti" il `JInternalFrame`.

`void addInternalFrameListener(InternalFrameListener l)`

Aggiunge a questo `JInternalFrame` un `InternalFrameListener`.

`void removeInternalFrameListener(InternalFrameListener l)`

Rimuove un `InternalFrameListener` da questo `JInternalFrame`.

Modello degli eventi di `JInternalFrame`

La gestione degli eventi è affidata ad oggetti di tipo `InternalFrameListener`, che

offrono un controllo totale degli eventi che possono capitare a un `JInternalFrame`.

Di seguito ecco i metodi dell'interfaccia `InternalFrameListener` con la descrizione del tipo di evento che ne provoca la chiamata.

```
void internalFrameActivated(InternalFrameEvent e)
```

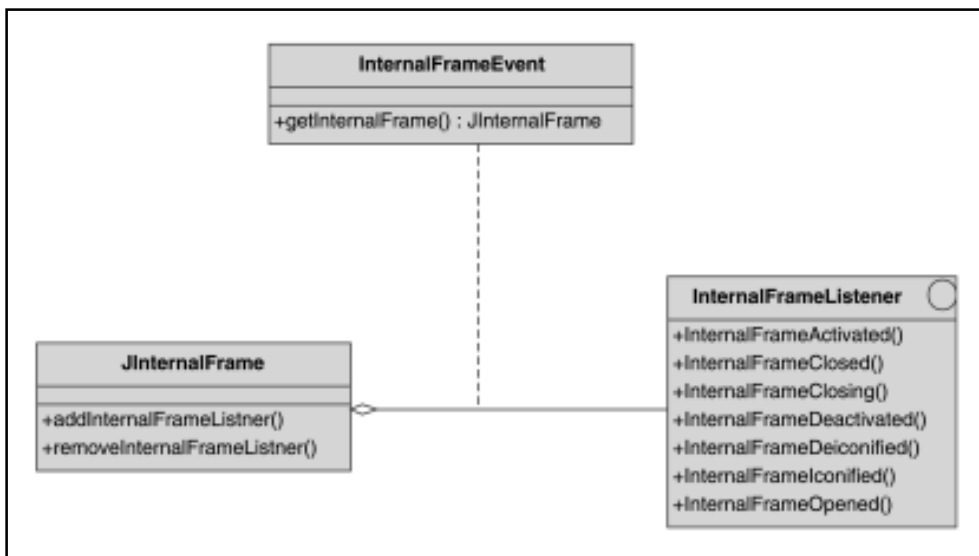
Un Internal Frame è stato attivato.

```
void internalFrameClosed(InternalFrameEvent e)
```

Un Internal Frame è stato chiuso.

```
void internalFrameClosing(InternalFrameEvent e)
```

Un Internal Frame sta per essere chiuso.

Figura 10.33 – *Il meccanismo di gestione degli eventi di* `JInternalFrame`

```
void internalFrameDeactivated(InternalFrameEvent e)
```

Un internal frame è stato disattivato.

```
void internalFrameDeiconified(InternalFrameEvent e)
```

Un internal frame è stato deiconificato.

```
void internalFrameIconified(InternalFrameEvent e)
```

Un Internal Frame è stato ridotto a icona.

```
void internalFrameOpened(InternalFrameEvent e)
```

Un Internal Frame è stato aperto.

Gli eventi di tipo `InternalFrameEvent` dispongono del pratico metodo `getInternalFrame()` che restituisce il `JInternalFrame` che ha generato l'evento.

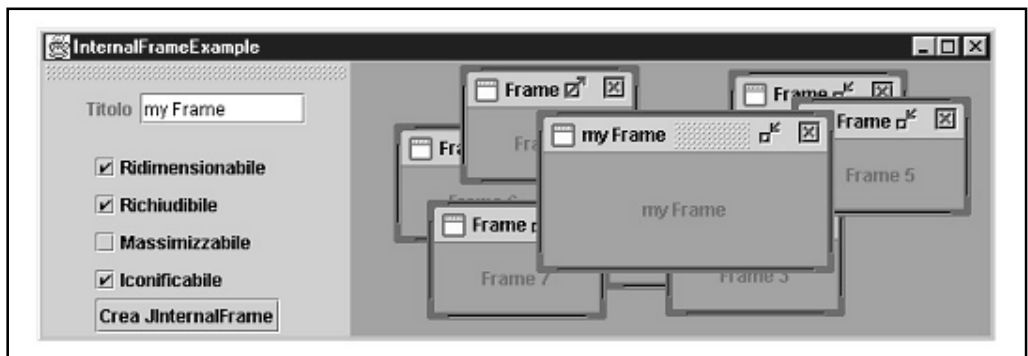
Un esempio pratico

Ecco ora un esempio completo: una fabbrica di `JInternalFrame`. Questo programma presenta un `JDesktopPane` e una `JToolBar`, nella quale sono presenti i controlli che permettono di creare dei `JInternalFrame` dopo averne impostate le proprietà. La politica di posizionamento viene implementata con formule matematiche nel metodo `createFrame()`, che calcola le coordinate tenendo conto della dimensione del frame, del desktop e dalla posizione dell'ultima finestra visualizzata.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class InternalFrameExample extends JFrame {
   private JDesktopPane desktop;
   private int frameNumber = 0;
   private int xPos = 0;
   private int yPos = 0;
   private JTextField titleTextField;
   private JCheckBox resizableCheckBox;
   private JCheckBox closableCheckBox;
```

```
    private JCheckBox maximizableCheckBox;
    private JCheckBox iconifiableCheckBox;

    public InternalFrameExample() {
        super("InternalFrameExample");
        setSize(640, 210);
        JToolBar toolbar = createToolBar();
        desktop = new JDesktopPane();
        getContentPane().add(BorderLayout.WEST, toolbar);
        getContentPane().add(BorderLayout.CENTER, desktop);
        setVisible(true);
    }
    protected JToolBar createToolBar() {
        JToolBar tb = new JToolBar(JToolBar.VERTICAL);
        JPanel titlePanel = new JPanel();
        titlePanel.setLayout(new FlowLayout());
            JLabel titleLabel = new JLabel("Titolo");
            titleTextField = new JTextField("Frame 0", 10);
        titlePanel.add(titleLabel);
        titlePanel.add(titleTextField);
        resizableCheckBox = new JCheckBox("Ridimensionabile");
        closableCheckBox = new JCheckBox("Richiudibile");
        maximizableCheckBox = new JCheckBox("Massimizzabile");
        iconifiableCheckBox = new JCheckBox("Iconificabile");
        JButton generateButton = new JButton("Genera un JInternalFrame");

        ActionListener listener = new GenerateButtonActionListener();
        generateButton.addActionListener(listener);
        titleTextField.addActionListener(listener);

        tb.add(titlePanel);
        tb.add(resizableCheckBox);
        tb.add(closableCheckBox);
        tb.add(maximizableCheckBox);
        tb.add(iconifiableCheckBox);
        tb.add(generateButton);
        return tb;
    }
    protected JInternalFrame createFrame(String title, boolean resizable,
                                         boolean closable, boolean maximizable,
                                         boolean iconifiable) {
        // Crea il Frame secondo i parametri
        JInternalFrame frame = new JInternalFrame(title, resizable, closable,
                                                  maximizable, iconifiable);
        frame.setSize(120, 80);
        // Aggiunge una Label al suo interno
        JLabel titleLabel = new JLabel(title, JLabel.CENTER);
        frame.getContentPane().add(titleLabel);

        // Calcola la posizione del nuovo InternalFrame
```

```
      int xSize = desktop.getWidth() - frame.getWidth();
      int ySize = desktop.getHeight() - frame.getHeight();
      int xStep = desktop.getWidth() / 10;
      int yStep = desktop.getHeight() / 10;
      xPos = (xPos + xStep) % xSize;
      yPos = (yPos + yStep) % ySize;
      frame.setLocation(xPos, yPos);

      return frame;
   }
   class GenerateButtonActionListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         String title = titleTextField.getText();
         boolean resizable = resizableCheckBox.isSelected();
         boolean closable = closableCheckBox.isSelected();
         boolean maximizable = maximizableCheckBox.isSelected();
         boolean iconifiable = iconifiableCheckBox.isSelected();
         JInternalFrame frame = createFrame(title, resizable, closable,
                                            maximizable, iconifiable);
         // aggiunge al JDesktopPane
         desktop.add(frame);
         // lo mette in cima agli altri JInternalFrame
         frame.moveToFront();
         // lo rende visibile
         frame.setVisible(true);
         titleTextField.setText("Frame " + String.valueOf(frameNumber++));
      }
   }
   public static void main(String[] args) {
      InternalFrameExample frame = new InternalFrameExample();
   }
}
```

Figura 10.34 – *Una fabbrica di* `JInternalFrame`

Accessori e decorazioni

JOptionPane

La classe `JOptionPane` permette di realizzare facilmente finestre modali di input, di allarme o di scelta multipla, ossia quel genere di finestre che vengono utilizzate qualora sia necessario segnalare un malfunzionamento, o presentare all'utente un gruppo di scelte su come procedere nell'esecuzione di un programma. L'API `JOptionPane` mette a disposizione tre tipi di pannelli: Confirm Dialog, Input Dialog e Message Dialog: il primo tipo di pannello viene usato quando si deve chiedere all'utente di effettuare una scelta tra un gruppo di possibilità; il secondo torna utile quando si debba richiedere l'inserimento di una stringa di testo, mentre il terzo viene usato per informare l'utente di un evento. La classe `JOptionPane` fornisce un gruppo di metodi statici che permettono di creare facilmente questi pannelli ricorrendo a una sola riga di codice: ecco un un esempio di Confirm Dialog.

```
JOptionPane.showConfirmDialog(null, "Salvare le modifiche?");
```

e uno di *Message Dialog*

```
JOptionPane.showMessageDialog(null, "Questo programma ha eseguito un'operazione
                              non valida e sarà terminato...", "Errore",
                              JOptionPane.ERROR_MESSAGE);
```

Figura 10.35 – *Un pannello di conferma può essere d'aiuto per evitare guai…*

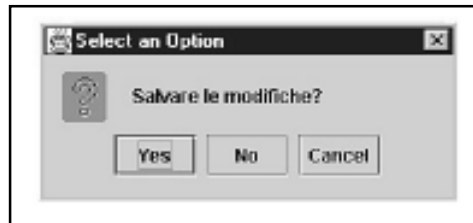
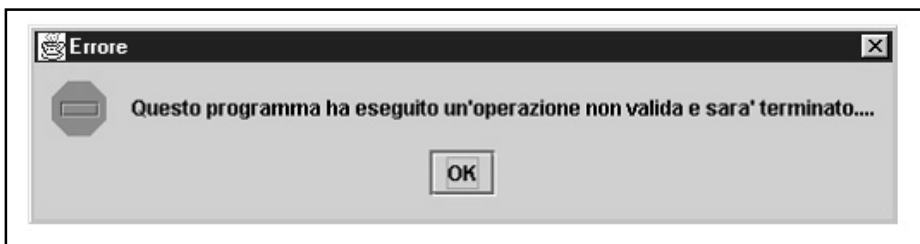



Figura 10.36 – *…un minaccioso pannello di notifica annuncia che è ormai troppo tardi*

Come si vede non è stato necessario creare esplicitamente alcun oggetto di tipo `JOptionPane`: in entrambi i casi è bastato richiamare un metodo statico che ha provveduto a creare un oggetto grafico con le caratteristiche specificate dai parametri. In questo paragrafo ci si concentrerà sull'utilizzo di un sottoinsieme di tali metodi, nella convinzione che essi permettano di risolvere la stragrande maggioranza delle situazioni in modo compatto ed elegante.

```
static int showConfirmDialog(Component parentComponent, Object message)
```

Mostra una finestra di conferma con le opzioni `Yes`, `No` e `Cancel` e con il titolo `Select an Option`. Il parametro permette di specificare il messaggio.

```
static int showConfirmDialog(Component parentComponent, Object message,
                             String title, int optionType, int messageType)
```

Mostra una finestra di conferma personalizzata.

```
static String showInputDialog(Component parentComponent, Object message)
```

Mostra una finestra di input generica.

```
static String showInputDialog(Component parentComponent, Object message,
                              String title, int messageType)
```

Mostra una finestra di input personalizzata.

```
static void showMessageDialog(Component parentComponent, Object message)
```

Mostra una finestra di informazione dal titolo `Message`.

```
static void showMessageDialog(Component parentComponent, Object message,
                              String title, int messageType)
```

Mostra una finestra di informazione personalizzata.

I metodi appena illustrati richiedono di specificare alcuni dei seguenti parametri:

`Parent Component`

Questo parametro serve a specificare il frame principale; esso verrà bloccato fino al termine dell'interazione. Ponendo a null questo parametro, la finestra verrà visualizzata al centro dello schermo e risulterà indipendente dal resto dell'applicazione.

`Message`

Questo campo permette di specificare una stringa da visualizzare come messaggio. In alternativa a `String` si può passare una `Icon` o una qualunque sottoclasse di `Component`.

`OptionType`

I Confirm Dialog possono presentare diversi gruppi di opzioni, e precisamente `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`.

`Message Type`

Attraverso questo parametro è possibile influenzare l'aspetto complessivo della finestra, per quanto attiene al tipo di icona, al titolo, al layout. Il parametro può assumere uno dei seguenti valori: `JoptionPane.ERROR_MESSAGE`, `JOptionPane.INFORMATION_MESSAGE`, `Joption-Pane.WARNING_MESSAGE`, `JOptionPane.QUESTION_MESSAGE`, `JOptionPane.PLAIN_MESSAGE`.

Le finestre create con i metodi `showConfirmDialog` e `showMessageDialog` restituiscono un intero che fornisce informazioni su quale scelta è stata effettuata dall'utente. Esso può assumere uno dei seguenti valori: `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION`, `JOptionPane.OK_OPTION`, `JOptionPane.CLOSED_OPTION`.

Nel caso di `showInputDialog` viene invece restituita una stringa di testo, o `null` se l'utente ha annullato l'operazione. Il programmatore può prendere decisioni su come proseguire leggendo e interpretando la risposta in maniera simile a come si vede in questo esempio:

```
int returnVal = JOptionPane.showConfirmDialog(null, "Salvare le modifiche?");
if(returnVal == JOptionPane.YES_OPTION)
   // procedura da eseguire in caso affermativo
else if(returnVal == JOptionPane.NO_OPTION)
   // procedura da eseguire in caso negativo
else;
   // operazione abortita
```

Figura 10.37 – *Un esempio di* `JFileChooser`

`JFileChooser`

Un file chooser è un oggetto grafico che permette di navigare il file system e di selezionare uno o più file su cui eseguire una determinata operazione. Qualunque applicazione grafica ne utilizza uno per facilitare le operazioni su disco. `JFileChooser` offre questa funzionalità attraverso una accessoriata finestra modale.

Si può creare un'istanza di `JFileChooser` utilizzando i seguenti costruttori

```
JFileChooser()
```

Crea un `JFileChooser` che punta alla home directory dell'utente.

```
JFileChooser(File currentDirectory)
```

Crea un `JFileChooser` che punta alla directory specificata dal parametro.

Per visualizzare un `JFileChooser` è possibile ricorrere alla seguente coppia di metodi, che restituiscono un intero.

```
int showOpenDialog(Component parent)
```

Visualizza un `JFileChooser` per apertura file.

```
int showSaveDialog(Component parent)
```

Visualizza un `JFileChooser` per salvataggio file.

L'intero che viene restituito può assumere uno dei tre valori `JFileChooser.CANCEL_OPTION`, `JFileChooser.APPROVE_OPTION`, `JFileChooser.ERROR_OPTION` e il programmatore può decidere cosa fare dei files selezionati basandosi su queste risposte.

Entrambi i metodi richiedono inoltre che venga passato come parametro un componente: di norma si passa un reference al `JFrame` principale, che verrà bloccato per tutta la durata dell'operazione. Passando `null` come parametro, il `JFrame` verrà visualizzato al centro dello schermo e risulterà indipendente dalle altre finestre. Per conoscere il risultato dell'interrogazione è possibile usare i seguenti metodi.

```
File getCurrentDirectory()
```

Restituisce la directory corrente.

```
File getSelectedFile()
```

Restituisce il file selezionato.

```
void setCurrentDirectory(File dir)
```

Imposta la directory di lavoro.

```
void setSelectedFile(File file)
```

Imposta il file selezionato.

Alcuni metodi permettono un uso più avanzato di `JFileChooser`.

```
void setDialogTitle(String dialogTitle)
```

Imposta il titolo del `JFileChooser`.

```
void setFileSelectionMode(int mode)
```

Permette di abilitare il `JFileChooser` a selezionare solo files, solo directory o entrambe, utilizzando come parametro uno dei seguenti valori: `JFileChooser.FILES_ONLY`, `JFileChooser.DIRECTORIES_ONLY`, `JFileChooser.FILES_AND_DIRECTORIES`.

```
void setMultiSelectionEnabled(boolean b)
```

Abilita o disabilita la possibilità di selezionare più di un file per volta. In questo caso, per interrogare lo stato, si ricorrerà ai due metodi che seguono.

```
void setSelectedFiles(File[] selectedFiles)
```

Imposta come selezionati il gruppo di files passati come parametro.

```
File[] getSelectedFiles()
```

Restituisce un vettore contenente i files selezionati dall'utente.

Nelle seguenti righe si può osservare una tipica procedura che fa uso di `JFileChooser`.

```
class MyFrame extends JFrame {
...
   fileChooser = new JFileChooser();
   int response = fileChooser.showOpenDialog(this);
   if(response == JFileChooser.APPROVE_OPTION) {
      File f = fileChooser.getSelectedFile();
      // qui viene eseguita l'operazione sul file
   }
...
}
```

Pluggable Look & Feel

Ogni ambiente a finestre è caratterizzato da due fondamentali proprietà: l'aspetto dei componenti (ovvero la loro sintassi), e la maniera in cui essi reagiscono alle azioni degli utenti (la loro semantica); l'insieme di queste proprietà viene comunemente definito Look & Feel.

Chiunque abbia provato a lavorare su un sistema Linux dopo anni di pratica su piattaforma Windows si sarà reso conto di quanto sia difficile abituarsi ad una nuova semantica:

le mani tendono a comportarsi come sulla vecchia piattaforma, ma la reazione che osserviamo con gli occhi non è quella che ci aspettavamo. Ad esempio su Linux è comune che le finestre si espandano verticalmente invece che a pieno schermo, o che i menù scompaiano quando si rilascia il tasto del mouse.

La natura multipiattaforma di Java ha spinto i progettisti di Swing a separare le problematiche di disegno grafico dei componenti da quelle inerenti al loro contenuto informativo, con la sorprendente conseguenza di permettere agli utenti di considerare il Look & Feel come una proprietà del componente da impostare a piacere.

La distribuzione standard del JDK comprende di base due alternative: Metal e Motif. La prima definisce un Look & Feel multipiattaforma, progettato per risultare il più possibile familiare a chiunque; la seconda implementa una vista familiare agli utenti Unix. Le distribuzioni di Java per Windows e Mac includono anche un L&F che richiama quello della piattaforma ospite; per motivi di copyright non è permesso alla Sun di proporre queste due scelte su piattaforme diverse. Alcune software house indipendenti distribuiscono, sotto forma di file JAR, dei package contenenti dei L&F alternativi: per aggiungerli alla lista dei L&F di sistema è sufficiente inserire questi files nel classpath.

Per impostare da programma un particolare Look & Feel è sufficiente chiamare il metodo `UIManager.setLookAndFeel(String className)` passando come parametro il nome di un L&F installato nel sistema, ad esempio:

```
try {
   UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
} catch (Exception e) { }
```

Di seguito si presentano le stringhe relative ai quattro L&F descritti sopra:

```
"javax.swing.plaf.metal.MetalLookAndFeel"
```

Specifica il Java Look & Feel, denominato Metal.

```
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
```

Specifica il Look & Feel Windows, disponibile solo su piattaforma Win32.

```
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
```

Permette di impostare un L&F in stile CDE/Motif. Questa scelta è disponibile su qualunque piattaforma.

```
"javax.swing.plaf.mac.MacLookAndFeel"
```

Definisce il Look & Feel Mac, disponibile solo su piattaforma Apple.

Se si desidera interrogare il sistema per conoscere il nome e la quantità dei L&F installati, si può ricorrere ai seguenti metodi statici di `UIManager`.

```
static String getSystemLookAndFeelClassName()
```

Restituisce il nome del `LookAndFeel` che implementa il sistema a finestre della piattaforma ospite (Windows su sistemi Microsoft, Mac su macchine Apple e Motif su piattaforma Solaris). Se non esiste una scelta predefinita, viene restituito il nome del Metal L&F.

```
static String getCrossPlatformLookAndFeelClassName()
```

Restituisce il nome del `LookAndFeel` multipiattaforma, il Java Look & Feel (JLF).

```
static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()
```

Restituisce un vettore di oggetti che forniscono alcune informazioni sui `LookAndFeel` installati nel sistema, tra le quali il nome (accessibile con il metodo `getName()`).

Border

Una caratteristica che Swing offre in esclusiva è la possibilità di assegnare un bordo diverso ad ogni singolo componente grafico, sia esso un pannello, un pulsante o una Tool Bar. Per aggiungere un bordo ad un componente, è sufficiente chiamare il metodo `setBorder(Border b)` passando come parametro un'istanza di una qualunque delle classi descritte di seguito.

Il package `javax.swing.border` offre ben sette tipi di bordo, il più semplice dei quali è composto da una singola riga, dello spessore specificato.

```
LineBorder(Color color, int thickness, boolean roundedCorners)
```

Crea un bordo a linea del colore, spessore e tipo di bordo specificato.

I seguenti bordi ricreano effetti tridimensionali: essi richiedono come parametro un intero che può assumere il valore `BevelBorder.LOWERED` o `BevelBorder.RAISED` a seconda che si desideri un effetto in rilievo o rientrante.

```
BevelBorder(int bevelType)
```

Crea un bordo in rilievo, del tipo specificato dal parametro.

```
SoftBevelBorder(int bevelType)
```

Crea un bordo in rilievo sfumato, del tipo specificato dal parametro.

```
EtchedBorder(int etchType)
```

Crea un bordo scolpito, del tipo specificato dal parametro.

Qualora si desideri creare attorno a un componente una vera e propria cornice di spessore arbitrario, è possibile ricorrere ai seguenti oggetti, che permettono creare bordi vuoti, a tinta unita o decorati con un'immagine GIF o JPEG.

```
EmptyBorder(int top, int left, int bottom, int right)
```

Crea un bordo vuoto dello spessore specificato.

```
MatteBorder(Icon tileIcon)
```

Crea un bordo utilizzando un'immagine.

```
MatteBorder(int top, int left, int bottom, int right, Icon tileIcon)
```

Crea un bordo delle dimensioni specificate utilizzando un'icona.

```
MatteBorder(int top, int left, int bottom, int right, Color matteColor)
```

Crea un bordo delle dimensioni e del colore specificati.

Per finire, è disponibile una coppia di bordi che permette di creare composizioni a partire da altri bordi

```
TitledBorder(Border border, String title, int titleJustification,
            int titlePosition, Font titleFont, Color titleColor)
```

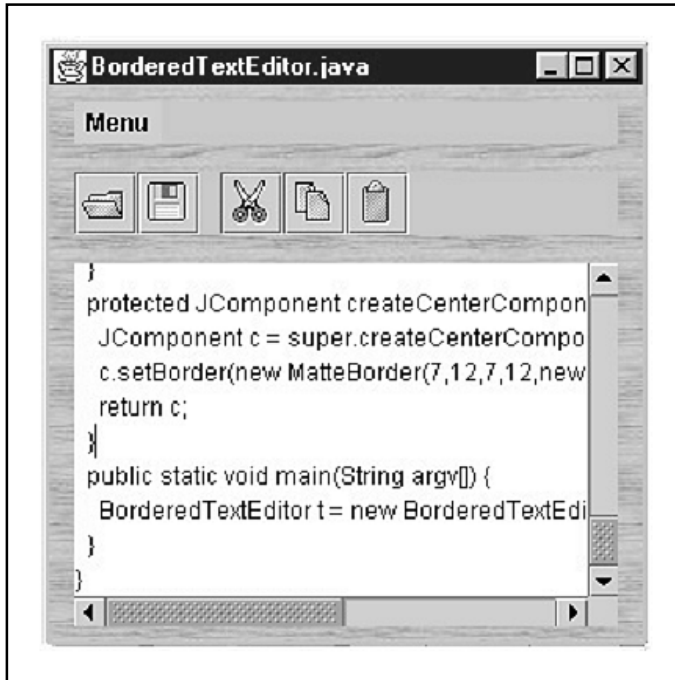
Crea una cornice composta dal bordo che viene passato come primo parametro e dal titolo specificato dal secondo parametro. Il terzo parametro può assumere i valori `TitledBorder.CENTER`, `TitledBorder.LEFT` o `TitledBorder.RIGHT`; il quarto, che specifica la posizione del titolo, può assumere invece i valori `TitledBorder.ABOVE_BOTTOM`, `TitledBorder.ABOVE_TOP`, `TitledBorder.BELOW_BOTTOM`, `TitledBorder.BELOW_TOP`. Gli ultimi due parametri specificano font e colore del titolo. Sono disponibili costruttori più semplici, ad esempio uno che richiede solo i primi due parametri ed uno che omette gli ultimi due.

```
CompoundBorder(Border outsideBorder, Border insideBorder)
```

Crea una cornice componendo i due bordi passati come parametro.

Il prossimo esempio illustra la costruzione di una sottoclasse di `TextEditor`, il programma descritto nella sezione relativa a `JTextArea`. Ridefinendo i Factory Methods è possibile modificare in maniera vistosa l'aspetto dell'applicazione, aggiungendo un bordo alla Menu Bar, alla Tool Bar e al pannello centrale, senza bisogno di alterare il costruttore del programma.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderedTextEditor extends TextEditor {
  protected JMenuBar createMenuBar() {
     JMenuBar mb = super.createMenuBar();
     mb.setBorder(new MatteBorder(7, 12, 7, 12,
                            new ImageIcon("Texture_wood_004.jpg")));
     mb.setBackground(new Color(224, 195, 96));
     return mb;
  }
     protected JToolBar createToolBar() {
        JToolBar tb = super.createToolBar();
        tb.setBorder(new MatteBorder(7, 12, 7, 12,
                               new ImageIcon("Texture_wood_004.jpg")));
        tb.setBackground(new Color(224, 195, 96));
        return tb;
     }
     protected JComponent createCenterComponent() {
        JComponent c = super.createCenterComponent();
        c.setBorder(new MatteBorder(7, 12, 7, 12,
                              new ImageIcon("Texture_wood_004.jpg")));
        return c;
     }
```

Figura 10.38 – *Aggiungendo i bordi è possibile dare un look più "elaborato" a qualunque applicazione*

```
   public static void main(String argv[]) {
      BorderedTextEditor t = new BorderedTextEditor();
   }
}
```

Conclusioni

In questo capitolo sono state affrontate tutte le tematiche di base della programmazione a finestre e sono stati introdotti alcuni argomenti avanzati: ora non resta che mettere in pratica quanto appreso. L'unico valido consiglio per chi desideri imparare a scrivere del buon codice grafico è di studiare un gran numero di programmi scritti da programmatori esperti, cercando di adattarli alle proprie esigenze, senza aver vergogna di "copiare". Una grossa fonte di ispirazione può provenire persino dai sorgenti delle API: nonostante le inevitabili difficoltà iniziali, è possibile arrivare a comprenderne il funzionamento fino ai più piccoli dettagli (in fondo si tratta pur sempre programmi Java). Anche senza arrivare

a tanto, esse costituiscono un modello di codice ben scritto e ben documentato. (È possibile trovare i sorgenti di tutte le API Java nel file `src.jar`, presente nella directory principale del JDK; per aprirlo basta lanciare il comando `jar xvf src.jar` da una console in quella stessa directory). Con il tempo e l'esperienza cresceranno la sicurezza e la capacità di trovare soluzioni originali alle infinite sfide che la programmazione grafica presenta allo sviluppatore.

Applet

DI STEFANO ROSSINI – LUCA DOZIO

Che cosa è una Applet?

Una Applet è un particolare programma Java inserito in una pagina HTML, scaricato dinamicamente dalla rete ed eseguito dalla Java Virtual Machine del browser.

Le Applet sono state di fondamentale importanza nella storia e nella diffusione di Java rappresentando il punto di forza di Sun per fare breccia nel mercato. Sfruttando l'esistenza di browser per le differenti piattaforme hardware e software Sun ha utilizzato le Applet per dimostrare, con successo, la portabilità Java. Le Applet hanno da questo punto di vista dato nuova vitalità ai web browser e in generale alle applicazioni web, rivalutandone le potenzialità.

Grazie al meccanismo del download automatico dalla rete, un'Applet può essere eseguita senza la necessità di installare software né sviluppare particolari client per le diverse piattaforme: la rete diviene un canale da cui prelevare il software, e il browser diventa il client che esegue un programma in locale.

L'inserimento all'interno della pagina HTML di un'Applet avviene per mezzo del tag `<APPLET>` con il quale tra l'altro si specifica il nome della classe Java che costituisce l'Applet stessa.

Attualmente tutti i browser incorporano una macchina virtuale e sono in grado di eseguire un'Applet. Purtroppo si riscontrano sostanziali differenze sia fra le varie marche di browser che fra le varie versioni della stessa casa produttrice (di Microsoft e Netscape le più diffuse). Queste differenti scelte politiche e tecnologiche delle aziende possono causare di fatto un comportamento diverso di un'Applet in esecuzione a seconda del browser utilizzato, ed introdurre difficoltà nello sfruttare le più recenti innovazioni del JDK. Ad esempio, per quanto riguarda la gestione dell'interfaccia grafica, proprio per garantire la massima portabilità, si è spesso costretti a effettuare una scelta conservativa limitandosi

all'utilizzo di AWT, a scapito delle più potenti Swing API per le quali è necessario avvalersi di un apposito plugin.

Al fine di garantire un elevato margine di sicurezza le Applet devono rispettare i vincoli imposti dal Security Manager della macchina virtuale del browser: in particolare un'Applet non può:

- accedere in nessun modo al file system della macchina locale;
- scrivere sul server da cui proviene;
- accedere ad host diversi da quelli di provenienza.

Questo tipo di scelta effettuata dai progettisti della Sun, per quanto possa sembrare fortemente restrittiva, ha lo scopo preciso di garantire un elevato livello di sicurezza. Con l'utilizzo di architetture a più livelli queste restrizioni perdono di importanza, e in quest'ottica l'Applet può essere sviluppata pensandola unicamente in veste di "puro" client.

Si tenga presente che già con l'introduzione del JDK 1.1, e ancora di più con Java 2, sono state messe a disposizione del programmatore delle tecniche avanzate per modificare la politica restrittiva del security manager tramite la firma delle Applet.

Lo scopo di questo capitolo è di affrontare i vari aspetti legati alla programmazione delle Applet, cercando di focalizzare l'attenzione sulla corretta modalità di progettazione e implementazione di applicazioni distribuite basate su Applet.

Differenze tra Applet e applicazioni

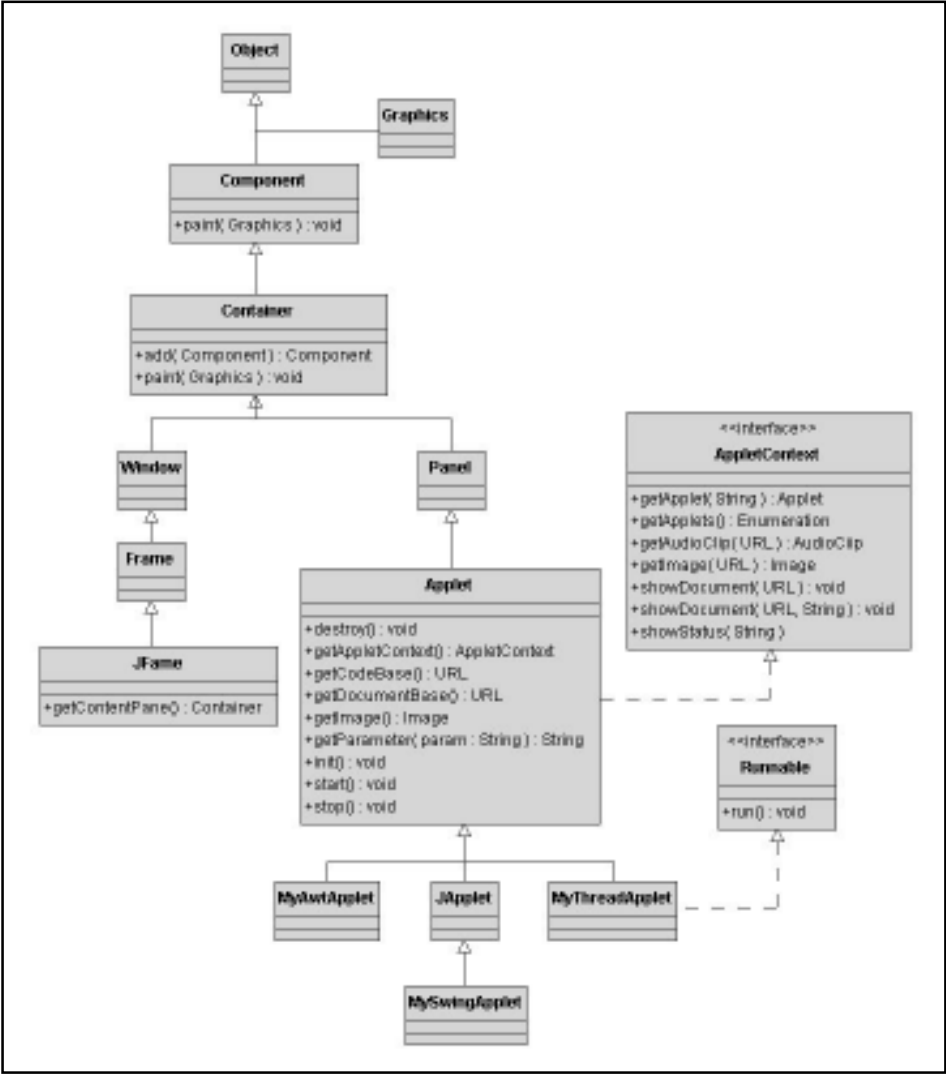
A causa del differente ambiente di esecuzione — la Virtual Machine del browser internet — le Applet presentano sostanziali differenze rispetto alle applicazioni Java standalone.

Per prima cosa, per creare un'Applet, si deve necessariamente estendere la classe `Applet` (o `JApplet` nel caso si tratti di un'Applet che usa la libreria grafica Swing), mentre un'applicazione standalone può essere dichiarata indipendentemente da altre classi.

L'Applet deve essere dichiarata `public`, perché la classe possa essere caricata e mandata in esecuzione. Se non si dichiara un'Applet `public` ma la si lascia senza specificare la tipologia di visibilità della classe (valore default, cioè la visibilità della classe è definita all'interno del package in cui è dichiarata), in fase di compilazione (compile time) non viene segnalato alcun errore, mentre durante l'esecuzione(run time) viene segnalato l'errore di "classe non trovata".

Per l’Applet l’inizializzazione avviene all’interno del metodo `init()`, dove tra le altre cose è possibile ricavare i parametri passati dall’esterno (vedi più avanti nel capitolo maggiori dettagli).

In fig. 11.1 si trova il diagramma UML, con indicate le classi e i metodi utilizzati negli esempi proposti nel capitolo.

Figura 11.1 – *Diagramma UML delle classi* `Applet` *e* `JApplet`

La prima semplice Applet: `HelloWorld.java`

Il primo esempio molto semplice è quello di un'Applet che visualizza il messaggio "HelloWorld!" a schermo. Il codice in esame è il seguente:

```
package com.mokabyte.mokabook.applets;

import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {

   /**
    * Messaggio che l'Applet visualizza
    */
   private String message;

   /**
    * Effettua le operazioni di inizializzazione dell'Applet
    */
   public void init() {
      message = "HelloWorld!";
   }

   /**
    * Disegna il Container dell'Applet
    * @param g oggetto da visualizzare
    */
   public void paint(Graphics g) {

   //drawRect:disegna un rettangolo delle dimensioni specificate
   g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);

   //drawString: disegna una stringa
   //nella posizione specificata dai valori dei parametri
   g.drawString(message, 5, 15);
   }
}
```

Come si può notare la visualizzazione del messaggio a video, ovvero nel contesto grafico dell'Applet, avviene ridefinendo il metodo `paint()`, e invocando i seguenti metodi sull'oggetto di classe `Graphics`:

`drawRect()`

Si occupa di disegnare un rettangolo delle dimensioni specificate dai valori che `getSize()` ritorna. Il metodo `getSize()` è un metodo della classe `Component`, che restituisce un oggetto di classe `Dimension`.

```
g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
```

drawString()

Si occupa di disegnare una stringa nella posizione specificata dai valori dei parametri x e y.

```
g.drawString(message, 5, 15); // x = 5, y = 15
```

Per poter eseguire l'Applet è necessario preparare un file HTML in cui sia inserito un riferimento tramite il tag CODE al file .class ottenuto al termine del processo di compilazione (in questo caso com.mokabook.applets.HelloWorld.class).

Di seguito è riportata una porzione di tale file in cui si effettua l'invocazione all'Applet nella sua forma più essenziale, garantendo l'aggiunta e la descrizione di ulteriori tag utilizzabili, quando se ne presenterà la necessità:

```
<APPLET CODE=com.mokabook.applets.HelloWorld.class
   WIDTH=400
   HEIGHT=300>
</APPLET>
```

I tag WIDTH e HEIGHT determinano le dimensioni dello spazio grafico dell'Applet, spazio che non è modificabile a runtime. I tag appena descritti sono gli unici indispensabili per il caricamento dell'Applet, e non possono essere tralasciati.

Lanciando da browser o mediante l'AppletViewer il caricamento della pagina HTML, dovrebbe venire visualizzata l'Applet con il messaggio HelloWorld!, in una finestra delle dimensioni dichiarate.

AppletViewer è un applicativo messo a disposizione nel JDK, che emula il browser nel caricamento delle Applet.

Se ciò non succede, due sono le possibili motivazioni:

- il browser non supporta Java e in tal caso vengono ignorati i tag sconosciuti e visualizzato il testo compreso tra <APPLET> e </APPLET>.

```
<APPLET CODE=com.mokabook.applets.HelloWorld.class
WIDTH=400 HEIGHT=300>
Attenzione! Il tuo browser non supporta le Applet.
</APPLET>
```

- il browser ha la Virtual Machine disabilitata; in questo caso l'Applet non potrà essere caricata e viene comunicato all'utente quanto sta accadendo mediante il messaggio specificato dal tag ALT.

```
<APPLET CODE=com.mokabook.applets.HelloWorld.class
WIDTH=400 HEIGHT=300
ALT="Attenzione! Il tuo browser ha la JVM disabilitata.">
</APPLET>
```

Passaggio di parametri: AppletWithParameter

A un'Applet è possibile passare parametri da pagina HTML. Infatti, racchiusi fra i tag <APPLET> e </APPLET>, si possono definire nomi (tag NAME) e valori (tag VALUE) di parametri che l'applicazione può leggere. Ad esempio

```
<APPLET CODE=com.mokabyte.mokabook.applets.AppletWithParameter.class
CODEBASE=classes
WIDTH=400
HEIGHT=300>
   <PARAM NAME=message VALUE="Ecco il valore che l'Applet passa...">
   <PARAM NAME=number VALUE=15>
</APPLET>
```

Rispetto all'esempio precedente è stato aggiunto il tag CODEBASE che indica al browser in quale directory è presente il bytecode dell'Applet.

Nell'esempio riportato il browser cercherà il file (appletWithParameter.class) nella directory

```
<ROOT_WEBSERVER>\applets\classes\com\mokabook\applets\
```

Ecco invece il codice dell'Applet: si noti come in questo caso, a differenza di quanto fatto in precedenza quando si erano disegnate le stringhe sovrascrivendo il metodo paint(), si sono utilizzati dei componenti grafici.

```
package com.mokabyte.mokabook.applets;

import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Label;

public class AppletWithParameter extends Applet {


   // Valore di testo in input dall'HTML
   private String message;
```

```
   // Valore di number in input da HTML
   private String number;

   //Label di visualizzazione di message
   private Label viewMessage;

   //Label di visualizzazione di number
   private Label numText;


   // Effettua le operazioni di inizializzazione dell'Applet
   public void init() {

      //getParameter: prende le informazioni da pag HTML
      message = getParameter("message");

      if(message==null)
      message = new String("Manca il tag HTML message");

      viewMessage = new Label(message);
      number = getParameter("number");

      if(number!=null){
         try {
            //...tentativo di conversione...
            int num = Integer.parseInt(number);
            //...conversione riuscita.
            numText = new Label(number);
         }
         catch(NumberFormatException e){
            //...conversione fallita: number non è un numero
            numText = new Label("Non è stato inserito un numero!");
         }
      }
      else{
         //Inizializzazione della Label
         numText = new Label("Manca il tag HTML number");
      }

      //Aggiunta di Label al Container dell'Applet
      add(viewMessage);
      add(numText);
   }
}
```

Dall'interno dell'Applet si può accedere ai parametri passati da file HTML utilizzando il metodo `getParameter(`*`nome del parametro HTML`*`)` della classe padre `Applet`.

Nell'esempio si legge il valore del parametro `message` e lo si memorizza nella proprietà `message`.

```
String message = getParameter("message")
```

È sempre bene, quando si prelevano parametri dall'esterno, assicurarsi della loro consistenza (valore diverso da `null`), in modo da evitare che sia lanciata una `NullPointerException` al momento dell'utilizzazione di tale variabile.

Inoltre si tenga presente che i valori dall'esterno sono sempre di tipo testuale, e quindi si dovranno prendere le opportune precauzioni (come utilizzare un blocco `try–catch`) nel momento in cui sia necessario effettuare conversioni verso tipi di dato strutturato.

Il codice HTML per l'Applet

La creazione di un'Applet non può prescindere dalla conoscenza dei tag HTML che ad

Tabella 11.1 – *Tag HTML per l'invocazione all'Applet*

tag	significato
`ALIGN` (opzionale)	Allineamento dell'Applet nella pagina HTML
`ALT` (opzionale)	Possibile messaggio di JVM non abilitata
`ARCHIVE` (opzionale)	Indica il file di archivio (estensione `.jar`) nel quale cercare le classi Java (file di estensione `.class`)
`CODEBASE` (opzionale)	Indica il percorso da webroot dove ricercare il file di estensione `.class` dell'applet
`CODE` (non opzionale)	Indica il file applet (estensione `.class`) da caricare
`NAME` (opzionale)	Assegna all'applet un nome referenziabile da JavaScript o usato nell'`appletContext` per la comunicazione fra applet nello stesso contesto
`HSPACE` (opzionale)	Definisce la spaziatura orizzontale.
`HEIGHT` (non opzionale)	Definisce l'altezza del canvas dell'applet che non può essere ridefinita a runtime. `Canvas` è una classe che implementa la classe astratta `Component` e che rappresenta un'area rettangolare dello schermo sulla quale l'applicazione può disegnare o recuperare eventi provocati dall'utente.
`VSPACE` (opzionale)	Definisce la spaziatura verticale
`PARAM` (opzionale)	Definisce i valori da passare all'Applet
`WIDTH` (non opzionale)	Definisce la larghezza del canvas dell'Applet che non può essere ridefinita a runtime

essa fanno riferimento. La tab. 11.1 è un quadro riassuntivo sul significato dei vari tag HTML utilizzati per gestire il comportamento delle Applet.

Il framework

Tipicamente un linguaggio implementa i meccanismi di base, i tipi di dato elementari, le strutture di controllo e demanda le operazioni effettive a specifiche librerie di classi.

I package Java sono collezioni di classi Java correlate tra loro, che il programmatore può utilizzare a piacimento all'interno dei propri programmi.

I package come AWT, Swing e Applet hanno un'impostazione molto diversa, dal momento che la sequenza delle chiamate ai metodi è predefinita: questo modello di implementazione, in cui il programmatore viene chiamato a estendere al fine di completare un'applicazione già definita, prende il nome di framework.

Quando si crea con il package Swing una finestra, senza scrivere nessuna riga di codice, si attiva un comportamento standard predefinito che è possibile specializzare ridefinendo degli opportuni metodi.

In sostanza l'approccio diviene per prima cosa dichiarativo, definendo il tipo di applicazione, Applet, Swing o AWT, per poi specializzarne il comportamento.

È importante tenere presente la differenza tra un framework e una comune libreria di classi, per quanto entrambi siano dei package. Le semplici collezioni di classi sono passive in quanto effettuano operazioni su richieste dei programmi, mentre i framework sono attivi e reagiscono in modo standard a determinati eventi esterni.

Prima di procedere allo sviluppo del framework relativo alle Applet è necessario conoscere il loro ciclo di vita, ovvero in corrispondenza di quali eventi il framework effettua in modo automatico la chiamata a opportuni metodi.

I metodi della classe `java.applet.Applet` che vengono invocati automaticamente dalla macchina virtuale del browser sono: `init()`, `start()`, `stop()` e `destroy()`.

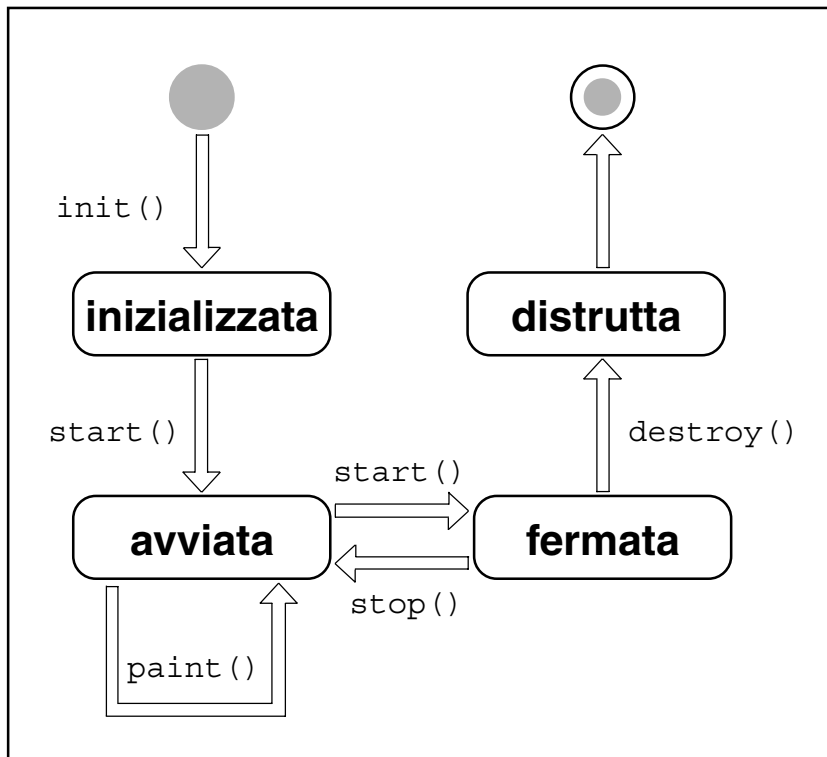
Ereditando dalla classe `Applet` e ridefinendo tali metodi è possibile specializzare una propria Applet.

Il metodo `init()`

Il metodo `init()` viene invocato una volta avvenuto il caricamento e la verifica del bytecode dell'Applet al fine di permettere la sua inizializzazione.

Ridefinendo tale metodo è possibile inserire il codice per effettuare la lettura dei parametri dichiarati dai tag HTML PARAM, caricare le risorse (immagini, suoni, file di testo, ecc.) e aggiungere componenti grafici per creare l'interfaccia grafica d'utente (GUI).

Il metodo effettua le operazioni che in un'applicazione standalone vengono normalmente eseguite nel costruttore. Il metodo `init()`, analogamente al costruttore, viene invocato una sola volta al caricamento della pagina HTML contenente l'Applet.

Figura 11.2 – *Ciclo di vita di un'Applet*

Il metodo `start()`

Viene invocato automaticamente dopo che la JVM ha invocato il metodo `init()` e ogni qualvolta l'Applet appare nel browser; ciò significa che il metodo `start()` può essere chiamato ripetutamente. Esso è il punto in cui solitamente viene avviato il thread dell'Applet, dove sono creati i thread aggiuntivi (ad esempio per effettuare animazioni o una qualsiasi operazione da eseguire in "parallelo" all'esecuzione dell'Applet) e dove sono effettuate tutte le operazioni chiuse dal metodo `stop()`.

Il metodo `stop()`

Viene invocato automaticamente quando l'utente esce dalla pagina in cui si trova l'Applet. Può quindi essere chiamato ripetutamente e ha lo scopo di consentire l'interruzione delle attività avviate nel metodo start evitando il rallentamento del sistema quando l'utente non sta visualizzando l'Applet.

Il metodo stop viene chiamato dal browser quando si cambia pagina senza però deallocare l'Applet.

Il metodo `destroy()`

Il metodo `destroy()` viene chiamato dopo che l'Applet è stata fermata mediante il metodo stop nel momento in cui il browser viene chiuso. In tale metodo va inserito il codice per rilasciare le risorse.

Il metodo `paint()`

Questo metodo, definito nella classe `Container`, permette di decorare l'area occupata dall'Applet all'interno della pagina HTML. Il metodo `paint()` è chiamato automaticamente quando l'Applet ha necessità di aggiornare il suo stato grafico, ad esempio in seguito alla sua visualizzazione sullo schermo per la prima volta, o perché qualche altra finestra si è sovrapposta a quella del browser o perché la stessa è stata ridimensionata. A fronte di tali eventi l'Applet chiama il metodo `update()` (ereditato dalla classe `Container`) che effettua l'operazione di aggiornamento tra le quali c'è anche la chiamata al metodo `paint()`. Il ciclo di ridisegnamento è il seguente:

- La chiamata di `repaint()` provoca la chiamata del metodo `update()`.
- Il metodo `update()` cancella il contenuto dello schermo (in sostanza lo ricopre con il colore di sfondo corrente) e richiama `paint()`.
- Il metodo `paint()` riproduce il fotogramma corrente.

La chiamata di `repaint()` (e quindi di `paint()`) non ha effetto immediato: essa è piuttosto una richiesta di ridisegnare l'Applet non appena possibile.

Tabella 11.2 – *Descrizione dei metodi invocati nel ciclo di vita di un'Applet*

metodo	invocazione	operazione
`init()`	Al caricamento dell'Applet	Inizializzazione dell'Applet
`start()`	Dopo il metodo `init()` e ad ogni visualizzazione della pagina HTML	Avvio o riavvio di thread
`stop()`	Prima di `destroy` e ogni volta che si cambia pagina	Arresto di thread
`destroy()`	Alla chiusura del browser	Rilascio delle risorse

Nel caso di operazioni cicliche il fatto che il metodo `update()` cancelli il contenuto dello schermo per poi ridisegnarlo provoca un fastidioso effetto di "sfarfallio" (*flickering*) sullo schermo.

Per eliminare tale effetto si può ridefinire il metodo `update()` all'interno del quale si chiama direttamente il metodo `paint()` evitando la pulizia del video.

Altro espediente per eliminare il flickering è quello della tecnica del double buffering che comporta il disegnare su una superficie grafica "fuori schermo" e successivamente copiare tale superficie direttamente sullo schermo.

I browser e il ciclo di vita delle Applet

Sempre a causa delle differenze presenti fra le varie implementazioni dei vari browser di cui si è parlato in precedenza, il ciclo di vita dell'Applet non è unico per tutti i browser. Lo stesso codice può produrre effetti diversi a seconda del browser che esegue l'Applet.

L'esempio dell'Applet `LifeCycle`, che viene riportato di seguito, permette di comprendere a fondo il ciclo di vita dell'Applet: in questo esempio, ad ogni invocazione di un metodo, si incrementa uno specifico contatore e si stampa il suo valore sulla Java console del browser. Il metodo `paint()` visualizza il valore corrente di tutti i contatori nel contesto grafico del browser.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.TextArea;
import java.awt.GridLayout;
import java.util.Date;

public class LifeCycle extends Applet {

   // Contatore di chiamate al metodo init
   static int initCounter = 0;
   // Contatore di chiamate al metodo start
   static int startCounter = 0;
   // Contatore di chiamate al metodo stop
   static int stopCounter = 0;
   // Contatore di chiamate al metodo destroy
   static int destroyCounter = 0;
   // Contatore di chiamate al metodo paint
   static int paintCounter = 0;

   // Effettua le operazioni di inizializzazione dell'Applet
   public void init() {
      addItem("init[" + (++initCounter) + "] - Thread["
               + Thread.currentThread().getName() + "]");
```

```
    }

    // Avvio dell'applet
    public void start() {
        addItem("start[" + (++startCounter) + "] - Thread["
                + Thread.currentThread().getName() + "]");
    }

    // Arresta l'esecuzione dell'Applet
    public void stop() {
        addItem("stop[" + (++stopCounter) + "] - Thread["
                + Thread.currentThread().getName() + "]");
    }

    //Effettua le operazioni di rilascio delle risorse
    public void destroy() {
        addItem("destroy[" + (++destroyCounter)+ "] - Thread["
                + Thread.currentThread().getName() + "]");
    }

    // Disegna il Container dell'Applet
    public void paint(Graphics g) {
        addItem("paint[" + (++paintCounter) + "] - Thread["
                + Thread.currentThread().getName() + "]");
        g.drawString("init[" + initCounter+ "] - start["
                      + startCounter + "] - paint[" + paintCounter
                      + "] - stop[" + stopCounter + "] -destroy["
                      + destroyCounter + "]", 100, 100);
    }

    // Scrive sulla Java console del browser
    void addItem(String newWord) {
        System.out.println(new java.util.Date().toString() + "-" + newWord);
    }
}
```

Per visualizzare la Java Console di Netscape bisogna selezionare dal menu la voce `Tasks` → `Tools` → `JavaConsole`.

Nel caso di Internet Explorer deve essere dapprima abilitata accedendo a `Strumenti` → `opzioni Internet` → `avanzate` selezionando la casella `Console Java attivata`.

Successivamente selezionare dal menu la voce `Visualizza` → `ConsoleJava`.

L'Applet `LifeCycle` è stata provata su quattro browser differenti: Internet Explorer 5.0 e 5.5 e Netscape 4.75 e 6.0. Osservando il valore dei contatori prodotti dall'Applet sulla console Java, si possono osservare differenze tra un ambiente di esecuzione e l'altro.

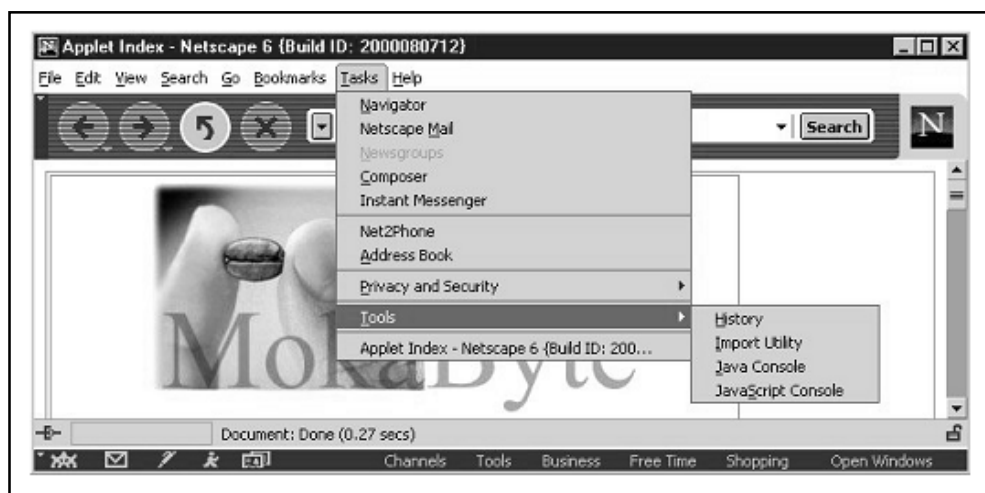
La tab. 11.3 riassume i risultati delle prove effettuate.

Tabella 11.3 – *Ciclo di vita per diversi browser a confronto*

	Internet Explorer		Netscape	
	5.0	**5.5**	**4.75**	**6.0**
Caricamento pagina HTML contenente l'applet	init start paint	init start paint	init start paint	init start stop
Caricamento di una nuova pagina HTML	stop destroy	stop destroy	stop	stop
Ritorno alla pagina HTML contenente l'applet	init start paint	init start paint	init start paint	start paint
Iconizzazione pagina	–	–	–	–
Da iconizzata a normale	paint	paint	paint	paint
Ridimensionamento finestra	paint	paint	paint stop start	paint
Sovrapposizione di un'altra finestra	paint	paint	paint	paint
Aggiornamento	stop destroy init start paint	stop destroy init start paint	stop destroy init start paint	stop start paint
Chiusura del browser	stop destroy	stop destroy	stop destroy	stop destroy

Figura 11.3 – *Netscape 6.0: la Java Console*

Figura 11.4 – *Internet Explorer: attivazione della Java Console*

Figura 11.5 – *Internet Explorer: visualizzazione della Java Console*

Il metodo `main` nelle Applet

L'Applet non deve obbligatoriamente definire il metodo `main` come in una normale applicazione Java perché il suo entry-point è costituito dal metodo `init`.

Nonostante questo è utile scrivere il metodo `main` all'interno della Applet per almeno due ragioni:

- permette di effettuare il test dell'Applet senza ricorrere all'AppletViewer o al browser;
- la sua presenza permette il funzionamento del programma anche come applicazione standalone.

Benché si utilizzino regole diverse per creare Applet e applicazioni, esse non entrano in conflitto tra loro, dato che il lifecycle delle Applet è ignorato quando il programma viene eseguito come applicazione.

L'Applet eseguita come applicazione avrà il metodo `main` che si sostituisce di fatto al motore del framework scandendo le invocazioni dei metodi.

Per gestire parti di codice specifiche per le Applet o per le applicazioni si può ricorrere al seguente espediente: si aggiunge una proprietà `isStandalone` di tipo boolean alla classe e si imposta il suo valore a `true` all'interno del metodo `main()` ove altrimenti vale `false`.

In questo modo il codice specifico per le Applet verrà eseguito nel caso in cui tale variabile sia `false`. Nell'esempio seguente, l'Applet viene visualizzata all'interno di un oggetto di classe `Frame` nel caso di funzionamento come applicazione.

```
public class MyApplet extends Applet {

   boolean isStandalone = false;
   ...
   public static void main(String[] args) {
      MyApplet applet = new MyApplet();
      applet.isStandalone = true;
      Frame frame = new Frame("MyApplet");
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      });
   frame.add(applet);
   frame.setSize(200, 200);
   applet.init();
   frame.setVisible(true);
}
```

Nel caso di Applet Swing, queste sono inserite in un oggetto di classe `JFrame`.

```
JFrame frame = new JFrame("MySwingApplet");
frame.getContentPane().add(applet);
```

Un esempio completo di programma Java utilizzabile sia come Applet che come applicazione standalone è riportato nel seguito del capitolo.

AppletContext

Un'Applet può chiedere al browser di eseguire una serie di operazioni particolari come riprodurre un audioclip, visualizzare un breve messaggio nella barra di stato, visualizzare un'altra pagina web, ecc.

Queste operazioni sono eseguite mediante l'utilizzo dell'interfaccia `AppletContext` la cui implementazione può essere considerata come un canale di comunicazione tra l'Applet e il browser locale.

Il metodo `getAppletContext()` della classe `Applet` restituisce un reference all'`AppletContext` tramite il quale si possono invocare i metodi riportati in tab. 11.4.

Nell'ambiente del browser si distinguono due aree: la barra di stato e l'area di visualizzazione della pagina Web. Un'Applet può accedere a entrambe mediante

Tabella 11.4 – *Metodi dell'interfaccia* `AppletContext`.

metodo	descrizione
`void showStatus(String message)`	Mostra la stringa "`message`" nella riga di stato del browser
`Image getImage (URL url)`	Restituisce un'immagine specificata nell'URL se esiste; altrimenti ritorna `null`
`AudioClip getAudioClip(URL url)`	Restituisce un oggetto `AudioClip` il quale memorizza il file audio specificato dall'URL
`Enumeration getApplets()`	Restituisce una `Enumeration` di tutte le Applet appartenenti allo stesso contesto
`Applet getApplet(String appletName)`	Restituisce l'Applet nel contesto corrente avente il nome specificato o `null` in caso contrario
`void showDocument(URL url)`	Visualizza una nuova pagina web
`void showDocument(URL url, String target)`	Visualizza una nuova pagina web in un frame nel browser nella modalità specificata dal parametro `target`

l'interfaccia `AppletContext`. Utilizzando il metodo `showStatus()` si può visualizzare una stringa sulla barra di stato del browser:

```
getAppletContext().showStatus("caricamento immagini");
```

Dato che questa barra è utilizzata dal browser per segnalare le operazioni in corso, le scritte dell'Applet possono essere sovrascritte da messaggi del tipo `Loading Java`..., `applet running` e così via.

Sfruttando il metodo `showDocument()` è possibile richiamare la visualizzazione di un'altra pagina web e di fatto accedere al suo contesto.

```
URL newUrl = new URL("http://www.mokabyte.it");
GetAppletContext().showDocument(newUrl);
```

Il metodo così invocato visualizza una nuova pagina sulla stessa finestra della pagina corrente, soppiantando così l'Applet in esecuzione.

Nella chiamata al metodo `showDocument()` è possibile specificare un secondo parametro, che indica al browser la modalità di visualizzazione della nuova pagina. La tab. 11.5 riporta gli argomenti del metodo `showDocument()`.

L'`AppletContext` è di fatto il contesto in cui "vive" l'Applet, delineando (e di conseguenza delimitando) le sue operazioni all'interno del browser che la ospita.

Comunicazione tra Applet nello stesso contesto

Se una pagina web contiene due o più Applet aventi lo stesso codebase, queste possono comunicare tra loro.

Assegnando al tag `NAME` di ciascuna Applet nel file HTML un nome, è possibile utilizzare il metodo `getApplet(String)` dell'interfaccia `AppletContext` per ottenere un riferimento all'Applet.

Tabella 11.5 – *Argomenti target del metodo* `showDocument`

target	posizione della nuova pagina aperta
`"_self"` o nessun parametro	Nel frame corrente
`"_parent"`	Nel contenitore parent
`"_top"`	Nel frame più alto
`"_blank"`	In una finestra nuova senza nome
Qualsiasi altra stringa	Nel frame con il nome specificato. Se non esiste alcun frame con quel nome, apre una nuova finestra e le assegna quel nome

Per esempio, se il valore di `NAME` di `MyApplet` nel file HTML è `GraveDigger`

```
<!-- Scrittura del tag NAME per specificare il nome dell'Applet -->

<APPLET code = "MyApplet.class" WIDTH = 300 HEIGHT = 100
NAME = "GraveDigger"> </APPLET>
```

l'Applet si rende disponibile verso le altre Applet della pagina. Il metodo `getApplet()` restituisce un riferimento all'Applet

```
Applet appletRef = getAppletContext().getApplet("GraveDigger");
```

tramite il quale è possibile invocare i metodi dell'Applet una volta effettuato il cast appropriato.

```
((MyApplet)appletRef).metodo(dati);
```

È possibile anche elencare tutte le Applet contenute in una pagina HTML dotate o meno di un tag `NAME`, utilizzando il metodo `getApplets()` che restituisce un oggetto di classe `Enumeration`.

```
Enumeration e = getAppletContext().getApplets();
while(e.hasMoreElements()) {
   Object a = e.nextElement();
   System.out.println(a.getClass().getName());
}
```

Un'Applet non può comunicare con Applet di pagine web diverse perché non appartengono al medesimo contesto. Il metodo `getApplets()` restituisce solo le Applet appartenenti allo stesso contesto.

`MyAppletContext` (riportata negli esempi allegati) permette di provare i metodi dell'interfaccia `AppletContext`. In particolare essa permette di visualizzare un messaggio sulla barra di stato del browser tramite l'istruzione

```
ac.showStatus(txtMessage.getText());
```

e di caricare una nuova pagina HTML specificando URL ed eventuali modalità grazie alla seguente porzione di codice

```
url = new URL(urlBase, txtNextPage.getText());
if(choice.getSelectedIndex() == 0)
   ac.showDocument(url);
else
```

```
ac.showDocument(url, choice.getSelectedItem());
```

È possibile inoltre ottenere l'elenco delle Applet del contesto

```
Enumeration enumAppletsOnThisPage = ac.getApplets();
```

Si può ottenere un riferimento ad un'altra Applet

```
Applet applet = ac.getApplet(txtAppletName.getText());
```

Questi ultimi due punti permettono di verificare che Applet in frame diversi, o in pagine diverse, non appartengano allo stesso contesto.

Comunicazione tra Applet in contesti differenti

Per permettere la comunicazione fra Applet appartenenti a contesti differenti e superare il limite imposto dai metodi `getApplets()` e `getApplet()`, si può sfruttare il fatto che la Virtual Machine del browser è condivisa tra le pagine HTML della medesima sessione. Si può quindi creare una classe con ad esempio una proprietà statica, con metodi di accesso statici, che memorizzi i reference delle Applet in esecuzione in pagine differenti al fine di renderli utilizzabili da tutte le Applet.

Tale classe (esempio `AppletLister.java`) svolge il ruolo di "ponte" tra le Applet in contesti diversi.

```
public class AppletLister {

   // Classe per memorizzare i reference delle Applet
   private static Hashtable applets = new Hashtable();

   // Metodo per registrare l'Applet
   public static void register(String name, Applet applet) {
      applets.put(name, applet);
   }

   // Metodo di rimozione dell'Applet
   public static void remove(String name) {
      applets.remove(name);
   }

   // Metodo che ritorna l'applet-reference
   // corrispondente al nome richiesto
   public static Applet lookupByName(String name) {
      return (Applet)applets.get(name);
```

```
   }

   // Metodo che ritorna un'Enumeration di
   // tutti gli applet-reference
   public static Enumeration getApplets() {
      return applets.elements();
   }
}
```

In questo caso per permettere la comunicazione tra Applet in contesti diversi si devono effettuare le seguenti operazioni:

- l'Applet "ricevitore" si deve registrare presso l'`AppletLister` mediante il metodo `register()` passando come parametri il suo nome e il suo riferimento.

```
AppletLister.register("AppletReceiver",this);
```

- l'Applet "trasmettitore" deve ottenere il reference di classe `java.applet.applet` invocando il metodo `lookupByName()` passando come parametro il nome dell'Applet con cui vuole comunicare.

```
receiver = getAppletContext().lookupByName("AppletReceiver");
```

- Se il reference ritornato è diverso da `null`, vuole dire che esiste un'Applet con quel nome e si può procedere a effettuare l'opportuno downcast e la relativa invocazione del metodo per la comunicazione.

```
if (receiver != null) {
   // controllo RTTI
if (receiver instanceof AppletReceiver)
   // Cast
((AppletReceiver)receiver).processRequest(myAppletName);
```

La sicurezza

Il modello di funzionamento di base di un'Applet, noto sotto il nome di network computing porta ad un processo di installazione ed esecuzione sulla macchina locale di codice remoto e di fatto sconosciuto. Per questo motivo molto lavoro è stato fatto al fine di fornire sufficienti garanzie all'utilizzatore finale delle Applet in materia di sicurezza. Il progetto della JVM prevede un meccanismo di loading dinamico (a runtime), di gestione della sicurezza e di protezione da azioni potenzialmente pericolose o intrusive.

Il modello SandBox

Il modello più semplice di gestione della sicurezza delle Applet è quello detto *sandbox*: in questo caso esse vengono eseguite all'interno di un ambiente protetto in modo da impedirne l'accesso al sistema sottostante.

La sandbox è quindi una specie di contenitore dal quale le Applet non possono evadere: il controllo sulle operazioni permesse è effettuato dal cosiddetto Security Manager, il quale verifica il bytecode in fase di caricamento (Bytecode Verifier), e l'Applet in fase d'esecuzione (Security Manager).

Il Security Manager della macchina virtuale del browser ha il compito di lanciare un'eccezione di tipo `SecurityException` ogni qualvolta un'Applet cerca di violare una delle regole di accesso come ad esempio scrivere sul server da cui proviene o collegarsi a host diversi da quelli di provenienza. Il modello sandbox è il più vecchio essendo stato introdotto fin dal JDK 1.0.

Le azioni vietate per le Applet sono:

- accedere ad indirizzi arbitrari in memoria; limite intrinseco del linguaggio Java e del Bytecode Verifier;
- accedere al file system locale;
- utilizzare procedure native: non si possono invocare i metodi della classe `java.lang.Runtime` come `exec` o `exit`;
- leggere alcune delle proprietà di sistema della macchina sulla quale sono in esecuzione;
- definire alcuna proprietà di sistema;
- creare o modificare alcun thread o thread group che non si trovi nel thread group dell'Applet stessa;
- definire o utilizzare una nuova istanza di `ClassLoader`, `SecurityManager`, `ContentHandlerFactory`, `SocketImplFactory` o `URLStreamHandlerFactory`; devono utilizzare quelle già esistenti;
- accettare connessioni da una macchina client.

Per quanto riguarda il quarto punto, le proprietà che un'Applet può leggere sono riportate nella tab. 11.6.

Le proprietà proibite per evidenti motivi di sicurezza sono quelle riportate in tab. 11.7.

Tabella 11.6 – *Proprietà leggibili da un'Applet*

nome proprietà	significato	esempio Windows	esempio Unix
"file.separator"	Separatore di file	/	\
"path.separator"	Separatore di percorso	;	:
"java.class.version"	Versione di Java class		
"java.vendor"	Stringa specifica di Java vendor		
"java.vendor.url"	URL di Java vendor		
"java.version"	Numero versione Java		
"line.separator"	Separatore di linea	Carriage return + Line feed	Line feed
"os.name"	Nome del sistema operativo	Windows 95, Windows NT	SolarisHP-UX, Linux
"os.arch"	Architettura del sistema operativo	X86, i586	Sparc PA-RISC Linux

Tabella 11.7 – *Proprietà non leggibili da un'Applet*

nome proprietà	significato	esempio Windows	esempio Unix
java.class.path	Java classpath	.;D:\jdk1.3\lib\rt.jar	.:/usr/java1.1/classes
java.home	Directory di installazione Java	C:\programmi\jre\1.3	/usr/java1.1
user.dir	Directory su cui si sta lavorando	D:\aranino	/users/ale/
user.home	Home directory dell'operatore	C:\Windows	/usr/java1.1
user.name	Account name dell'operatore		

L'`AppletPropertiesReader` che segue è un esempio di programma Java che può essere utilizzato sia come applicazione standalone che come Applet e mostra le differenze di accesso alle proprietà di sistema.

Nel funzionamento standalone il programma può accedere a tutte le proprietà mediante il metodo `getProperties()` della classe `System`

```
Properties sysProp = System.getProperties();
```

mentre un'Applet deve specificare il nome della proprietà che vuole leggere mediante il metodo `getProperty()` della classe `System`.

Nell'esempio riportato i nomi delle proprietà sono inseriti nel vettore di stringhe di nome `propertyNames`.

```
value = System.getProperty(propertyNames[i]);
```

Ad ogni lettura di proprietà si controlla l'eventuale verificarsi di una `SecurityException`.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.util.*;

public class AppletPropertiesReader extends Applet implements Runnable {

   // Se vale true indica che il programma
   // è una applicazione standalone
   private boolean isStandalone = false;

   // Vettore dei nomi delle proprietà da
   // leggere dall'Applet
   private String[] propertyNames = {  "browser", "browser.version",
                                       "file.separator", "line.separator",
                                       "path.separator", "java.class.version",
                                       "java.vendor", "java.vendor.url",
                                       "java.version", "os.name",
                                       "os.arch", "os.version" };

   Label[] values;
   Label[] names;

   // Effettua le operazioni di inizializzazione dell'Applet
   public void init() {

      if(isStandalone) {
         System.out.println("Applicazione standalone:
                        lettura di tutte le proprietà di sistema");
         try {
            int i=0;
            // leggo tutte le properties di sistema
            Properties sysProp = System.getProperties();
            // le stampo su console
            sysProp.list(System.out);
            Enumeration enum = sysProp.propertyNames();
            // assegno al reference propertyNames un
```

```
            // nuovo array di stringhe
            propertyNames = new String[sysProp.size()];
            // carico nell'array gli elementi
            // contenuti nell'Enumeration
            while(enum.hasMoreElements())
               propertyNames[i++] = (String)enum.nextElement;
      }
         catch (SecurityException e) {
            System.out.println("Could not read: " + e.getMessage());
         }
      }

      buildGUI();
      new Thread(this, "Loading System Properties").start();
   }

   // Costruisce l'interfaccia grafica d'utente
   private void buildGUI() {
      String firstValue = "not read yet";

      names = new Label[propertyNames.length];
      values = new Label[propertyNames.length];

      for (int i = 0; i < propertyNames.length; i++) {
         names[i] = new Label(propertyNames[i]);
         add(names[i]);
         values[i] = new Label(firstValue);
         add(values[i]);
      }
      this.setLayout(new GridLayout(propertyNames.length,2));
      this.setVisible(true);

      Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
      setSize(dim.width/2, dim.height/3);
   }

   // Corpo del thread: lettura delle proprietà
   public void run() {
      String value = null;
      Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

      //Attesa di 2 secondi per visualizzare i valori di default
      try {
         Thread.sleep(2000);
      }
      catch (InterruptedException ie) {
         System.out.println("InterruptedException " + ie.getMessage());
      }
```

```
      for (int i = 0; i < propertyNames.length; i++) {
         try {
            Thread.sleep(200);
         }
         catch (InterruptedException ie) {
            System.out.println("InterruptedException" + ie.getMessage());
      }


      try {
         value = System.getProperty(propertyNames[i]);
         values[i].setText(value);
      }
      catch (SecurityException se) {
         values[i].setText("SecurityException");
         System.out.println("SecurityException: propietà NON leggibile."
                            + se.getMessage());
         }
      }
   }

   // Punto di partenza dell'applicazione standalone
   public static void main(String args[]) {
      System.out.println("Metodo main invocato: funzionamento standalone...");
      AppletPropertiesReader applet = new AppletPropertiesReader();
      applet.isStandalone = true;
      Frame frame = new Frame("MyApplet");
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      });
      frame.add(applet);
      frame.setSize(300, 300);
      applet.init();
      applet.start();
      frame.setVisible(true);
   }
}
```

Si è messo in risalto che un'Applet può prelevare file soltanto dall'host di provenienza. Per non incorrere in errori durante l'esecuzione di operazioni di rete è importante utilizzare i metodi `getDocumentBase()` o `getCodeBase()`: il primo recupera l'URL della pagina contenente l'Applet mentre il metodo `getCodeBase()` recupera l'URL della directory contenente gli Applet.

L' `AppletGetBase` mostra il funzionamento di questi due metodi.

```
package com.mokabook.applets;

import java.awt.*;
import java.applet.*;
import java.net.*;

public class AppletGetBase extends Applet {

   private static final int NUM_ROWS = 14;

   private TextField txtDocumentBase = new TextField();
   private TextField txtCodeBase = new TextField();
   private TextField txtHostFromDocBase = new TextField();
   private TextField txtHostFromCodeBase = new TextField();
   private TextField txtHostPort = new TextField();
   private TextField txtHostName = new TextField();
   private TextField txtHostAddress = new TextField();

   // Effettua le operazioni di inizializzazione dell'Applet
   public void init() {

   // Document base in formato stringa
   txtDocumentBase.setText(getDocumentBase().toString());

   // Code base in formato stringa
   txtCodeBase.setText(getCodeBase().toString());

   // Ottengo l'host dato il document base
   txtHostFromDocBase.setText(getDocumentBase().getHost());

   // Ottengo l'host dato il code base
   txtHostFromCodeBase.setText(getDocumentBase().getHost());

   // La porta di ascolto del Web Server
   txtHostPort.setText(" " + getDocumentBase().getPort());

   // Ricava l'indirizzo IP dell'host
   try {
      InetAddress iaHost = InetAddress.getByName(txtHostFromDocBase.getText());
      System.out.println("hostName = " + iaHost.getHostName());
      System.out.println("hostAddress="+ iaHost.getHostAddress());
      txtHostName.setText(iaHost.getHostName());
      txtHostAddress.setText(iaHost.getHostAddress());
   }
   catch(Exception e) {
      System.out.println(" # Exception = " + e.getMessage());
   }

   this.setLayout(new GridLayout(NUM_ROWS, 1, 10, 5));
```

```
    add(new Label("Document base: "));
    add(txtDocumentBase);
    add(new Label("Code base: "));
    add(txtCodeBase);
    add(new Label("Host from Document base: "));
    add(txtHostFromDocBase);
    add(new Label("Host from Code base: "));
    add(txtHostFromCodeBase);
    add(new Label("HostPort: "));
    add(txtHostPort);
    add(new Label("Host name: "));
    add(txtHostName);
    add(new Label("Host address: "));
    add(txtHostAddress);
    this.setVisible(true);
    }
}
```

Figura 11.6 – *Uso di* `getDocumentBase()` *e* `getCodeBase()` *per recuperare l'host*

In fig. 11.6 si nota come reperire il nome dell'host utilizzando `getDocumentBase().getHost()` e `getCodeBase().getHost()` sia equivalente.
Si riportano alcuni esempi significativi sul modo in cui usare i due metodi.

Scaricare file di immagini:

```
Image img = getImage(getCodeBase(), "images/" + "figura.jpg");
```

Scaricare file audio:

```
AudioClip audioClip = getAudioClip(getCodeBase(), "audio.mid");
```

Connessione TCP/IP:

```
socket = new Socket(getCodeBase().getHost(), 1666);
```

Connessione RMI:

```
MyServerInterface serverRef
= (MyServerInterface)Naming.lookup("//" + getDocumentBase().getHost()
                                   + "/MyService");
```

Le Applet firmate

Oltre al classico modello sandbox, con l'avvento della versione 1.1 del JDK si è voluta offrire al programmatore una maggior libertà d'azione, introducendo il concetto di Applet firmata. In questo caso, tramite un processo di firma elettronica con un certificato digitale, si permette al security manager del browser di abbassare il livello di guardia e di garantire comunque all'utilizzatore la provenienza della Applet stessa. In questo modo, solo se si è certi della non pericolosità del codice scaricato, si può permettere all'Applet un maggior campo d'azione.

La firma delle Applet e il relativo processo di installazione sono piuttosto complessi, tanto che sempre più raramente vengono utilizzati, dando spazio a soluzioni in cui una differente architettura sia in grado di offrire le medesime potenzialità.

Come prima cosa si può prendere in esame la `AppletRebel`, un'Applet che tenta di ribellarsi alle regole imposte dal modello sandbox cercando di leggere la proprietà `user.name` e il file `auotexec.bat`. Come è lecito aspettarsi entrambe le operazioni generano una `SecurityException`.

La prima cosa da fare per permettere questo tipo di operazioni all'Applet è definire un set di regole di policy: tali regole devono essere definite secondo una sintassi particolare all'interno di un file situato nella directory `<JAVA_HOME>\JRE\lib\security`. Ad esempio le seguenti modifiche offrono alla `AppletRebel` il margine operativo richiesto.

```
grant {
   permission java.io.FilePermission "C:\AUTOEXEC.BAT", "read";
   permission java.util.PropertyPermission "user.name", "read";
};
```

Per forzare l'utilizzo di tali regole è necessario specificare l'utilizzo del file di policy ad esempio lanciando con l'Applet Viewer l'`AppletRebel` nel seguente modo

```
appletviewer -J-Djava.security.policy
= myfile.policy http://manowar:8000/html/AppletRebel.html
```

Specificando il file di policy si permette a tutte le Applet di seguire le regole in esso definite: un passo ulteriore consiste nell'imporre solo alle Applet fidate, cosiddette trusted o signed, la possibilità di eseguire le regole di policy.

Perché un'Applet sia fidata deve essere provvista di firma digitale: il JDK mette a disposizione a tal proposito degli strumenti in grado di firmare le Applet e di definire la security policy specificando le permission presenti in un certo policy file.

Per poter firmare le Applet queste devono essere incluse in un file JAR, che va opportunamente referenziato nella pagina HTML.

```
<applet code = "AppletRebel.class"
archive = "appletrebel.jar"
width  =400 height = 400>
<param name = file value = "c:\autoexec.bat">
</applet>
```

Di seguito sono riportate passo passo le operazioni necessarie per firmare l'`AppletRebel`.

- Compilazione del sorgente `AppletRebel.java`.

  ```
  javac -d . appletRebel.java
  ```

- Creazione del file JAR `appletrebel.jar`. In questo caso i flag `-cvf` permettono rispettivamente la creazione di un nuovo archivio, la modalità verbose e l'inserimento del nome dell'archivio da creare.

  ```
  jar -cvf appletrebel.jar .\com\mokabook\applets\*.class
  ```

- Generazione delle chiavi pubblica e privata. A questo punto, è necessario generare due chiavi asimmetriche che saranno utilizzate rispettivamente per firmare l'Applet (privata) e per consentire alla macchina client di verificare la firma digitale (pubblica). Quest'ultima sarà distribuita attraverso un certificato reso disponibile a tutti gli utilizzatori finali dell'Applet. Il comando per effettuare questa operazione è il seguente:

  ```
  keytool -genkey -alias "Me" -keystore "MyStore"
  ```

  L'opzione `-genkey` permette la generazione di una coppia di chiavi identificata attraverso l'alias `"Me"`.

- Memorizzazione della coppia di chiavi generata in un *keystore* (un file criptato) di nome `MyStore` che si trova nella directory dal quale è stato lanciato il comando. Durante la sua esecuzione, se non specificato all'invocazione, il keytool richiede alcune informazioni aggiuntive: password (`"MyPassword"`), nome e cognome, nome organizzazione, città, provincia, stato, nazione; key password per l'alias `"Me"` (`abc123`). Tali informazioni servono al keytool per associare un nome distintivo X.500 all'alias e possono essere specificate mediante linea di comando con l'opzione `-dname` del keytool. Il comando `-genkey` genera automaticamente un certificato autofirmato X.509 per la coppia di chiavi contenute nel file di nome `mystore`. Ottenute queste chiavi, esse possono essere utilizzate per firme successive di Applet differenti, oppure si può decidere di generarne di nuove. Una volta ottenute le chiavi si procede alla firma della Applet.

- Firma dell'archivio. Attraverso il tool `jarsigner`, si può procedere alla firma dell'archivio che contiene l'Applet.

  ```
  jarsigner -keystore MyStore
  -storepass MyPassword -keypass abc123 appletrebel.jar Me
  ```

- Verifica del procedimento di firma. Prima di proseguire è importante verificare che le operazioni avvenute siano andate a buon fine mediante l'opzione `-verify`.

  ```
  jarsigner -verify appletrebel.jar
  ```

  Se l'operazione di firma è andata a buon fine viene visualizzato il messaggio `jar verified` altrimenti `jar is unsigned`.

- Export del certificato. Per poter distribuire la chiave pubblica agli utenti finali che la utilizzeranno, si effettua generazione del certificato (export) nel seguente modo:

  ```
  keytool -export -keystore MyStore -alias Me -file MyCert.cer
  ```

fornendo la keystore password (`"MyPassword"`). In questo caso, il certificato viene memorizzato nel file `MyCert.cer`.

Quelle viste fino a qui sono le operazioni necessarie per firmare l'Applet e distribuire attraverso il certificato la chiave pubblica agli utenti finali dell'applicazione.

L'operazione che segue deve essere effettuata sulla macchina client per importare il certificato:

```
keytool -import -keystore YourStore -alias You -file MyCert.cer
```

Tale operazione richiede l'inserimento di una keystore password per il file `YourStore`. In questo modo, il certificato viene importato nel keystore `YourStore` con alias `You`. Dopo aver importato il certificato, in un policy file e per ognuna delle risorse che saranno utilizzate, verranno definite le permission di accesso delle Applet firmate da uno specifico signer.

```
keystore "YourStore";
grant SignedBy "You" {
   permission java.io.FilePermission "c:\AUTOEXEC.BAT", "read";
   permission java.util.PropertyPermission "user.name", "read";
};
```

Con il policy file riportato, viene data la possibilità alle sole Applet firmate da `"You"` di leggere il file `autoexec.bat` e la proprietà `user.name`.

```
appletviewer -J-Djava.security.policy
= myfile_2.policy http://manowar:8000/html/AppletRebel.html
```

Nel caso in cui l'esecuzione dell'Applet debba avvenire all'interno di un browser, le operazioni di import del certificato e di definizione delle permission sulle risorse del sistema dovranno essere effettuate con le funzionalità messe a disposizione dal browser medesimo.

Le Applet e l'interfaccia grafica: da AWT a Swing

La scelta fatta per il package AWT, cioè quella di voler creare una libreria grafica che fosse un massimo comune denominatore fra le librerie grafiche dei vari sistemi operativi, si prefiggeva di realizzare un ambiente grafico conforme a quello della piattaforma ospite. Il fatto di essere pioniera in tal senso è la causa prima che rende AWT una libreria piuttosto povera, con interfacce grafiche scarne e con un numero di funzionalità limitato.

Per aumentare le potenzialità espressive del linguaggio, la progettazione della libreria Swing è stata affrontata ripensando al modello implementativo e rifacendosi al noto Model

View Controller, che nelle Swing diventa modello per delega. L'interfaccia Swing risulta così leggera (*lightweight*), in grado di gestire un maggior numero di eventi e indipendente dall'implementazione delle classi grafiche del sistema operativo ospitante (componenti peer).

La realizzazione di Applet che utilizzano la libreria grafica Swing introduce nuove problematiche all'atto del caricamento dell'applicazione. Per poter dare una completa visione di questo tipo di problematiche si può riconsiderare l'`AppletWithParameter` presentata in precedenza, rivedendola in chiave Swing.

```
package com.mokabyte.mokabook.applets;

import javax.swing.JApplet;
import javax.swing.JLabel;

public class AppletWithParameterSwing extends JApplet {

   private String message;
   private String number;
   private JLabel viewMessage;
   private JLabel numText;
   public void init() {

   message = getParameter("message");
      if(message == null)
         message= "Manca il tag HTML message");
      viewMessage = new JLabel(message);
      number = getParameter("number");
      if(number != null) {
         try {
            int num = Integer.parseInt(number);
            numText = new JLabel(number);
         }
         catch(NumberFormatException e) {
            numText = new Jlabel("Non è stato inserito un numero");
         }
      }
      else {
         numText = new JLabel("Manca il tag HTML number");
      }

      //aggiunti componenti al Container di alto livello
      this.getContentPane().add("Center", viewMessage);
      this.getContentPane().add("South", numText);

   }

}   // End AppletWithParameterSwing
```

L'Applet in questione non estende più dalla classe `Applet` ma dalla `JApplet` (fig. 11.1) che fa parte del package `javax.swing`. Per quel che riguarda la costruzione dell'interfaccia grafica, l'aggiunta di componenti non viene più fatta direttamente sul contenitore dell'Applet

```
add(<componente grafico>);
```

ma sul `ContentPane`, un pannello di contenimento accessibile attraverso il metodo `getContentPane()`, delegato a contenere i componenti grafici.

```
this.getContentPane().contentPane.add(<componente grafico>);
```

equivalente a

```
Container contentPane = this.getContentPane();
contentPane.add(<Componente grafico>);
```

Caricando una pagina HTML contenente un'Applet Swing si riscontrano alcuni problemi. I messaggi di malfunzionamento variano a seconda del browser utilizzato.

L'analisi dei messaggi (tab. 11.8) induce a pensare a una situazione d'errore differente. Da un'osservazione più attenta, che prevede la visualizzazione del messaggio d'errore sulla Java Console, si scopre che il messaggio di IE sulla status bar è fuorviante.

Infatti il motivo per cui non viene eseguita l'Applet Swing è che la JVM dei browser non supporta Java 2 e questo genera un'impossibilità di caricamento della classe `Japplet` del package Swing. Il motivo di tutto ciò è che, al momento, solamente Netscape 6 supporta Java 2, nonostante sia già da qualche tempo che la libreria è stata rilasciata dalla Sun. Attualmente esistono diverse modalità di intervento per rendere il proprio browser "Java 2 Enabled".

Prima della versione 6 del browser Netscape, tutte le applicazioni di questo tipo prevedevano una Virtual Machine inserita al loro interno e non sostituibile. Per questo motivo

Tabella 11.8 – *I messaggi d'errore restituiti da diversi browser*

browser	messaggio in StatusBar
Explorer 5.5	Load: class AppletWithParameterSwing not Found!
Explorer 5.0	Load: class AppletWithParameterSwing not Found!
Netscape 6	Nessun messaggio di errore
Netscape 4.75	applet AppletWithParameterSwing error:java.lang.NoClassDefFoundError: javax/swing/Japplet

le limitazioni dovute alla non compatibilità con una determinata versione del JDK non erano direttamente risolvibili. Sun ha per questo motivo introdotto il cosiddetto Java Plugin, anticipando di qualche anno quello che poi si è rivelato essere lo scenario attuale o futuro, in cui ogni browser potrà utilizzare una VM a scelta in modo custom. Questa soluzione estende il set di funzionalità utilizzabili a tutto ciò che è offerto dalla piattaforma Java 2: per questo si possono utilizzare le API dell'ultimo momento, come Servlet API, EJB, JINI e così via.

Java Plug-in™

Il Java Plug-in™ è un tool che permette al browser di utilizzare una Virtual Machine esterna e quindi di supportare versioni del JDK differenti da quella della JVM incapsulata nel browser stesso. Pur trattandosi di un JRE a tutti gli effetti, il Java Plug-in™ segue il funzionamento di ogni altro tipo di plugin che supporta il browser per particolari applicazioni: in questo caso, effettuando le necessarie modifiche al file HTML, il controllo dell'Applet passa dalla JVM del browser a quella del plugin in modo del tutto automatico.

Per quanto riguarda la fase di download e installazione del plugin è possibile utilizzare appositi tag HTML in modo da automatizzare queste fasi tramite semplici click del mouse sulle pagine visualizzate dal browser. Come molto spesso accade nel mondo HTML, purtroppo anche per il plugin è necessario creare versioni differenti del codice HTML a seconda del browser utilizzato. Ad esempio, nel caso di Netscape 4.7, si deve utilizzare il tag `<EMBED>` al posto del tag `<APPLET>`. Tale marcatore porta con sé una serie di informazioni sulla tipologia di plugin richiesto e, nel caso in cui questo non sia già installato sul client, sul relativo URL di download. Per i browser Microsoft invece il tag da utilizzare è `<OBJECT>`.

Riprendendo in esame l'esempio visto in precedenza relativo all'Applet `AppletWithParameterSwing`, si riconsideri il codice HTML necessario per eseguire tale Applet; nel caso in cui si debba utilizzare la JVM del browser, tale script potrebbe essere

```
<APPLET CODE=com.mokabook.applets.AppletWithParameterSwing.class
CODEBASE=..\applets\classes\
WIDTH=400
HEIGHT=300>
   <PARAM NAME=message VALUE="Ecco il valore che l'Applet passa...">
   <PARAM NAME=number VALUE=15>
</APPLET>
```

Volendo invece utilizzare il plugin con Internet Explorer si otterrà la seguente versione modificata

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
```

```
WIDTH="400"
HEIGHT="300"
CODEBASE
="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
   <PARAM NAME="TYPE" VALUE="application/x-java-applet;version=1.2">
<PARAM NAME="CODE"
VALUE="com.mokabook.applets.AppletWithParameterSwing.class">
   <PARAM NAME="CODEBASE" VALUE="..\applets\classes\">
   <PARAM NAME ="message" VALUE ="Ecco il valore che l'Applet passa...">
   <PARAM NAME ="number" VALUE =15>
</OBJECT>
```

Nell'intestazione dell'`<OBJECT>` è presente una serie di attributi che descrivono le caratteristiche del Java Plug-in oltre a quelli che specificano i valori dei `PARAM` dell'Applet:

- un `classid` per l'identificazione del controllo ActiveX, della lunghezza di 16 byte e del valore di `"8AD9C840-044E-11D1-B3E9-00805F499D93"`. Questo numero indica che sarà scaricato un Java Plug-in;
- una serie di attributi dell'Applet (`HEIGHT`, `WIDTH`);
- un attributo `CODEBASE`, da **non** confondere con il `CODEBASE` dell'Applet, che specifica l'URL da dove deve essere scaricato il Java Plug-in. Il `CODEBASE` dell'Applet è specificato mediante il tag `<PARAM NAME="CODEBASE" VALUE ="...\applets\classes\">`;
- un numero di versione del Plug-in (`Version=1,2,0,0`).

Dopo il primo tag `<OBJECT>`, il browser si aspetta di trovare una serie di costrutti del tipo

```
<PARAM NAME="XXX" VALUE="XXX">
```

che sono usati per dichiarare al Java Plug-in i parametri che caratterizzano l'Applet. Ci si trovano dichiarati:

- in CODE, il file di estensione `.class`;
- in CODEBASE, il path relativo dove cercare il file `.class`;
- in TYPE, l'identificativo del tipo di file che si va ad eseguire.

Tabella 11.9 – *Tabella attributi*

nome di attributi/parametri	descrizione
`classid` (attributo)	Attributo non opzionale che identifica il controllo ActiveX per il Java Plug-in
`codebase` (attributo)	Attributo non opzionale che specifica l'URL del sito da cui scaricare il Java Plug-in. L'indirizzo specificato è quello del sito della Sun's Java Software Division Web
`height` (attributo)	Attributo non opzionale che specifica l'altezza del componente (in questo caso una Applet)
`width` (attributo)	Attributo non opzionale che specifica la larghezza del componente (in questo caso una Applet)
`archive` (parametro)	Parametro opzionale che specifica il nome del file JAR.
`code` (parametro)	Parametro non opzionale che specifica il nome dell'Applet o del componente
`codebase` (parametro)	Parametro opzionale che specifica il direttorio nel quale si deve trovare il file `.class` o il file `.jar`
`object` (parametro)	Parametro non opzionale che specifica il nome serializzato di un'Applet o di un componente
`type` (parametro)	Parametro non opzionale che identifica la tipologia del file da eseguire (p.e.: una Applet o un componente)

La tab. 11.9 è un quadro riassuntivo con tutti i parametri che entrano in gioco nell'utilizzo del tag `<OBJECT>`.

La versione modificata invece per il browser Netscape potrebbe essere la seguente

```
<EMBED
   TYPE="application/x-java-applet;version=1.2"
   CODE="com.mokabook.applets.AppletWithParameterSwing.class"
   CODEBASE="=..\applets\classes\"
   WIDTH=400
   HEIGHT=300
   PLUGINSPAGE="http://java.sun.com/products/plugin/1.2/plugin-install.html"
   message="Ecco il valore che l'Applet passa..."
   number="15">
</EMBED>
<NOEMBED>
   Non è stato trovato il supporto per il JDK 1.2
</NOEMBED>
```

In corrispondenza di questi tag, Netscape riconosce una coppia di tag `<NOEMBED>` e `</NOEMBED>` che si occupano di avvisare l'utente nel caso si dovesse presentare qualche

errore. Il tag `<EMBED>` contiene una serie di attributi che vengono usati dal Java Plug-in. `CODE`, `CODEBASE`, `HEIGHT`, `WIDTH`, hanno l'identico significato dei parametri del tag `<APPLET>`. L'attributo `PLUGINSPAGE` specifica l'indirizzo del sito da cui scaricare il Java Plug-in; l'attributo `TYPE` informa il Plug-in sulla tipologia del file da eseguire (Applet o bean).

In questo caso i tag `PARAM` sono scomparsi e il passaggio di parametri all'Applet è fatto direttamente nell'intestazione della chiamata `<EMBED>`, cioè i parametri sono trattati come attributi. Vi si trova il valore del messaggio (`message`) e del numero (`number`), che l'`AppletWithParameterSwing` ha come parametri in input. Per non creare confusione fra nome di attributi effettivi e nome di parametri dell'Applet è consigliato utilizzare un prefisso `JAVA_` per gli attributi, mentre i parametri si lasciano invariati.

```
ARCHIVE          JAVA_ARCHIVE
CODE             JAVA_CODE
OBJECT           JAVA_OBJECT
CODEBASE         JAVA_CODEBASE
TYPE             JAVA_TYPE
```

La tab. 11.10 offre un quadro riassuntivo con il significato dei vari attributi.

È possibile scrivere del codice HTML che si occupa di gestire la richiesta di caricamento del Plugin in modo automatico a seconda del browser che si sta utilizzando. In quest'ottica risultano fondamentali le seguenti osservazioni

- Netscape Navigator ignora i tag `<OBJECT>` e `</OBJECT>`, e conseguentemente i vari `PARAM` corrispondenti;

- per Internet Explorer bisogna introdurre due nuovi tag `<COMMENT>` e `</COMMENT>`, che includono nella loro sezione la gestione del tag `<EMBED>`. Il browser Microsoft considera il codice compreso fra i tag semplice commento e non lo interpreta.

Tabella 11.10 – *Tabella attributi*

nome attributo	descrizione
`Archive`	Attributo opzionale che specifica il nome del file JAR
`Code`	Attributo non opzionale che specifica il nome dell'Applet
`Codebase`	Attributo opzionale che specifica la directory contenente i file `.class` o `.jar`
`Height`	Attributo non opzionale che specifica l'altezza dell'Applet.
`Pluginspage`	Attributo non opzionale che specifica l'indirizzo Internet del sito da cui scaricare il Java Plug-in
`Type`	Attributo non opzionale che identifica il tipo di eseguibile (Applet o Bean)
`Width`	Attributo non opzionale che specifica la larghezza della finestra del componente

Si ottiene quello che segue:

```
<!— Codice HTML per Explorer -->
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH="400"
HEIGHT="300"
CODEBASE
="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
   <PARAM NAME="TYPE" VALUE="application/x-java-applet;version=1.2">
<PARAM NAME="CODE"
VALUE="com.mokabook.applets.AppletWithParameterSwing.class">
   <PARAM NAME="CODEBASE" VALUE="..\applets\classes\">
   <PARAM NAME ="message" VALUE ="Ecco il valore che l'Applet passa...">
   <PARAM NAME ="number" VALUE =15>
<!— Codice HTML per Netscape -->
<COMMENT>
   <EMBED
TYPE="application/x-java-applet;version=1.2"
   CODE="com.mokabook.applets.AppletWithParameterSwing.class"
   CODEBASE="=..\applets\classes\"
   WIDTH=400
   HEIGHT=300
   PLUGINSPAGE="http://java.sun.com/products/plugin/1.2/plugin-install.html"
   message="Ecco il valore che l'Applet passa..."
   number="15">
   </EMBED>
   <NOEMBED>
   Non è stato trovato il supporto per il JDK 1.2
   </NOEMBED>
</COMMENT>
</OBJECT>
<!— Fine codice HTML per Explorer e per Netscape -->
```

Come si sarà sicuramente notato il codice HTML modificato per l'esecuzione del plugin Java è piuttosto complesso, soprattutto se si desidera mantenere la compatibilità con entrambi i browser. Sun ha messo a disposizione il tool HTMLConverter che consente di modificare automaticamente l'HTML partendo da quello originario: tale tool si può scaricare direttamente dal sito Sun.

Internazionalizzazione delle applicazioni

DI GIOVANNI PULITI

Introduzione

Con l'avvento di Internet il mondo ha ridotto notevolmente le sue dimensioni. Ogni pagina web è distante un solo click da una qualsiasi altra, offrendo da un lato un numero incredibile di possibilità, ma ponendo al contempo una serie di problematiche nuove.

Si pensi ad esempio alle applet e alle applicazioni web: in questo caso la possibilità di fare eseguire una applicazione in una qualsiasi parte del mondo introduce il problema legato alla lingua utilizzata nella descrizione dell'interfaccia grafica o alle regole grammaticali in uso in quella regione geografica.

Anche se il problema non è nuovo, con l'avvento della rete si deve tener conto del fattore tempo, dato che la medesima applicazione può essere eseguita nel medesimo istante in luoghi differenti.

Il concetto di portabilità non è più quindi esclusivamente legato ai dettagli tecnologici della piattaforma utilizzata, ma deve tener conto anche di tutti quei formalismi in uso nel Paese di esecuzione per la rappresentazione di testi, valute, date e numeri.

Nasce quindi l'esigenza di poter disporre di strumenti che offrano la possibilità di scrivere applicazioni in modo indipendente dalla cultura di coloro che le andranno a utilizzare.

Il processo volto alla creazione di programmi slegati dal particolare contesto linguistico e grammaticale in uso nel momento dell'utilizzo del software, è detto localizzazione e internazionalizzazione di una applicazione.

La localizzazione permette l'adattamento del software a una particolare regione aggiungendo all'applicazione tutte quelle parti che dipendono dalla regione stessa, come ad esempio le scritte di una interfaccia grafica.

L'internazionalizzazione invece consente a una applicazione di adattarsi in modo automatico alle convenzioni in uso in un particolare linguaggio o regione senza che si debba ricorrere alla riscrittura o, peggio, alla riprogettazione dell'applicazione stessa: si pensi in questo caso alle differenti modalità di rappresentazione dei numeri o delle date.

Dato che molte volte risulta piuttosto scomodo utilizzare per intero la parola internationalization (si pensi ad esempio ai nomi delle directory), essa viene spesso indicata con la sigla i18n, dato che nella parola internationalization, sono presenti 18 caratteri fra la prima lettera i e l'ultima lettera n. La localizzazione sempre per lo stesso motivo viene abbreviata con la sigla l10n.

Per brevità quando ci si riferisce alla internazionalizzazione si intendono entrambi gli aspetti, ed è anzi piuttosto raro sentire parlare di *localization*. Si tenga presente che nella maggiore parte dei casi la *internationalization* non rappresenta l'unica soluzione ai problemi derivanti dalla localizzazione, ma offre notevoli vantaggi in termini di praticità e semplificazione. Anche in questo caso non sono state aggiunte particolari innovazioni tecnologiche rispetto al passato, limitandosi invece a mettere ordine e contestualizzare il lavoro e l'esperienza fatta in passato con altri linguaggi e tecnologie.

L'interfaccia grafica internazionale

Nel caso in cui un determinato software debba soddisfare i requisiti di *locale-independent* la prima problematica che si deve cercare di risolvere è la traduzione automatica delle varie scritte e messaggi che compongono l'interfaccia grafica.

Si consideri ad esempio il caso della label di un pulsante di conferma in una finestra di dialogo: nei Paesi di lingua inglese potrà essere utilizzata l'etichetta "OK", composta da due caratteri, mentre se l'applicazione deve essere venduta in Italia, si dovrebbe utilizzare la stringa "CONFERMA", costituita invece da otto caratteri. In questo caso quindi, senza gli strumenti del package `java.text` si dovrebbe creare un pulsante per l'applicazione inglese ed uno per quella italiana con dimensioni diverse. Se inoltre il bottone è un componente bean allora la filosofia di base della riutilizzabilità viene meno.

Data entry

Una corretta interazione con l'utente finale deve tener conto anche della fase di inseri-

mento dati, dato che una fonte d'errori consiste nella errata valutazione del contesto esatto di frasi e dati in genere.

Le API relative all'Internationalization non risolvono completamente il problema, ma permettono di dividere in maniera decisa l'implementazione delle classi dal linguaggio che esse supportano. I vantaggi di questa scelta emergono quando si devono estendere tali classi o aggiungere linguaggi nuovi.

Un esempio tipico è dato dalla tastiera utilizzata: le tastiere italiane infatti sono diverse da quelle giapponesi, e in questo caso i caratteri a disposizione di un giapponese (con PC abilitato a tale lingua) sono molti di più di quelli a disposizione di un italiano. Inoltre in Giappone per visualizzare un carattere non è sufficiente premere un tasto ma occorre il più delle volte una loro combinazione.

Java, fin dalla prima versione, prevede la gestione del formato Unicode che contiene quasi tutti i caratteri utilizzati nel mondo e quindi il problema non è la disponibilità dei caratteri ma la conoscenza di quale font utilizzare. Quando Java inizializza lo stream di input (`System.in`) viene letto il file `font.properties` nella directory `\lib` relativa a quella del JDK.

Questo file organizzato secondo il formato dei file Properties, contiene le informazioni sul font utilizzato per l'input.

Nel caso dell'Italia l'ultima riga di tale file è la seguente

```
# charset for text input
#
inputtextcharset = ANSI_CHARSET
```

in cui si specifica che il set di caratteri utilizzato è quello ANSI. Nel caso del Giappone il file `font.properties` sarebbe stato quello che nel nostro Paese prende il nome `font.properties.ja`. In questo file l'ultima riga è la seguente

```
# charset for text input
#
inputtextcharset = SHIFTJIS_CHARSET
```

in cui si indica che il font per l'inserimento dei caratteri è quello giapponese.

Si tenga presente che per utilizzare caratteri giapponesi in un programma Java non è sufficiente modificare le ultime righe del file `font.properties` dato che anche il sistema operativo in uso deve essere abilitato a tale font.

Il problema legato al punto precedente è la varietà dei font intesi come caratteri e non come corrispondenza carattere–tasto.

Oltre la sintassi

Oltre alla corrispondenza dei caratteri, esistono problemi di corrispondenza nella strut-

tura di una medesima frase nell'ambito di due contesti linguistici differenti: si supponga ad esempio di dover stampare un messaggio funzione di una determinata variabile come nel caso seguente.

```
System.out.println("La torre è alta " + res + " metri");
```

Si potrebbe anche scrivere

```
System.out.println(res + " metri è l'altezza della torre");
```

Il problema è che non tutte le lingue seguono la struttura soggetto–predicato–complemento, ma possono avere strutture più complesse in cui, come nel caso del cinese o del turco, l'ordine delle parole nella frase è molto significativo.

La gestione della struttura della frase e dei messaggi utilizzati nell'applicazione è un tipo di compito più complesso e verrà visto nella parte finale del capitolo.

La gestione dei font

Java ha adottato da subito lo standard Unicode. Nel JDK 1.0 si utilizza lo Unicode 1.1 mentre nella versione 1.1 si utilizza lo Unicode 2.0 che prevede la gestione di ben 38 885 caratteri derivanti dalla unione di quelli utilizzati da 25 culture diverse: i caratteri utilizzati dalla lingua italiana sono compresi tra `\u0020` e `\u007E`. Lo Unicode però, pur essendo un formato universale, non sempre viene utilizzato per la codifica dei caratteri; a seconda del caso particolare, la preferenza viene accordata ad altri formati.

Un formato molto utilizzato ad esempio è l'UTF-8 (Uniform Text Format a 8 bit) che permette di rappresentare ciascun carattere Unicode in un numero variabile di byte a seconda del carattere da rappresentare, essendo questo un codice a lunghezza variabile. Alcuni caratteri sono rappresentati con un unico byte, altri con due e altri con tre. Quelli rappresentati con un unico byte coincidono con quelli ASCII. Ciascun byte utilizza i primi bit come identificatori della sua posizione all'interno della codifica del carattere.

L'UTF-8 viene spesso utilizzato come formato di codifica delle stringhe durante le trasmissioni dato che, tra l'altro, è compatibile sui sistemi Unix.

Utilizzando un editor ASCII, come quello utilizzato ad esempio dalla maggior parte degli ambienti di sviluppo, per inserire un carattere Unicode non ASCII è necessario ricorrere alla sua codifica.

Per convertire una stringa da formato Unicode a UTF-8 si può operare come nel programma `Utf2Unicode`: supponendo di voler rappresentare la stringa `"perché"`, si può scrivere

```
String unicodeString = new String("perch " + "\u00E9");
```

Allora tramite il metodo `getBytes()`, specificando il formato è possibile ricavare un array di caratteri in formato UTF-8; ad esempio

```
byte[] utf8Bytes = original.getBytes("UTF8");
```

mentre

```
byte[] unicodeBytes = original.getBytes();
```

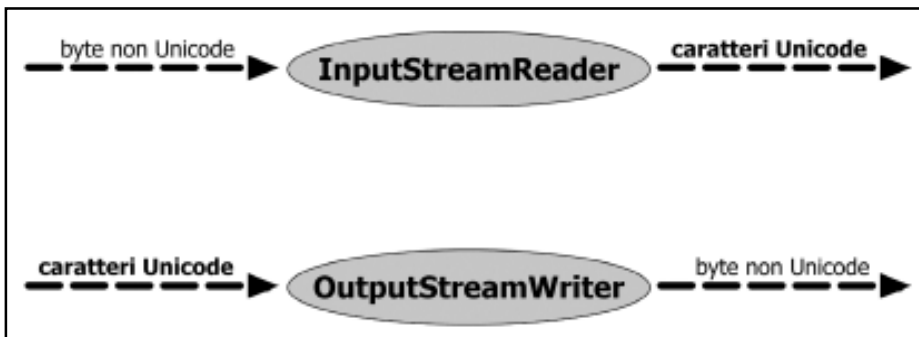
permette di ricavare un array analogo, ma contenente caratteri in formato Unicode.

Effettuando la stampa di entrambi gli array ci si potrà rendere conto effettivamente di come l'UTF-8 sia un formato a codifica variabile.

Per riottenere una stringa Unicode partendo dal formato UTF-8 è sufficiente specificare tale informazione nel costruttore della `String`, ovvero

```
String unicodeString2 = new String(utf8Bytes, "UTF8");
```

Si è detto che Java gestisce l'Unicode internamente trasformando ogni carattere proprio della cultura locale nel corrispondente carattere Unicode. Nella versione JDK 1.0 non era possibile agire su questa conversione mentre dalla versione JDK 1.1 è possibile farlo indirettamente attraverso le due classi `InputStreamReader` e `OutputStreamWriter` contenute nel package `java.io`.

Figura 12.1 – *Processo di conversione degli stream di caratteri da non Unicode a Unicode e viceversa*

Queste due classi, facendo uso di due particolari classi del package `sun.io` ottengono la conversione da Byte a Char e viceversa, in maniera *locale sensitive* cioè utilizzando i caratteri locali.

Tali classi devono poter disporre di un meccanismo per conoscere la regione in cui sono in esecuzione in modo da produrre la giusta conversione e infatti utilizzano la proprietà `file.encoding` delle Properties di sistema.

Nella tabella che segue sono riportati alcuni dei valori possibili (gli altri sono disponibili nella documentazione del JDK 1.1).

Tabella 12.1 – *Descrizione della codifica del* `file.encoding`

proprietà `file.encoding`	**descrizione della codifica**
8859_1	ISO Latin-1
8859_2	ISO Latin-2
8859-3	ISO Latin-3
Cp1250	Windows Eastern Europe / Latin-2
Cp1251	Windows Cyrillic
Cp1252	Windows Western Europe / Latin-1
Cp866	PC Russian
MacThai	Macintosh Thai
UTF8	Standard UTF-8

Nella classe `ProvaFileEncoding` viene brevemente mostrato come utilizzare tali classi per la conversione. Il cuore di questo semplice programma è il seguente:

```
// Esegue la conversione
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
InputStreamReader isr = new InputStreamReader(pis, txf_encoding.getText());
BufferedReader br = new BufferedReader(isr);
OutputStreamWriter osr = new OutputStreamWriter(pos, "8859_1")
BufferedWriter bw = new BufferedWriter(osr);

// Scrive sul writer
int lunghezza = txf_testo.getText().length();
char[] caratteri = new char[lunghezza];
txf_testo.getText().getChars(0, lunghezza, caratteri, 0);
bw.write(caratteri);
bw.flush();

// Legge il valore immesso da tastiera
br.read(caratteri, 0, lunghezza -1);
br.close();
bw.close();

// Visualizza il valore
lab_output.setText(new String(caratteri));
```

Osservandone i risultati si può notare come i caratteri accentati non sempre hanno una corrispondenza nel font del risultato. Per verificarlo basta provare una codifica di tipo `Cp1251` che corrisponde al Windows Cyrillic.

Infine si tenga presente che — poiché lo standard Unicode rappresenta ciascun carattere con due byte ovvero 16 bit, considerando però che molti caratteri che si utilizzano in un programma Java possono essere rappresentati attraverso un solo byte, e dato inoltre che molti calcolatori utilizzano una rappresentazione interna dei caratteri a otto bit — è stato reso disponibile un meccanismo per la rappresentazione dei caratteri Unicode in byte (anche per mantenere alta la portabilità del codice) attraverso i metodi

```
public final String readUTF() throws IOException

public static final String readUTF(DataInput in) throws IOException

della classe DataInputStream e

public final void writeUTF(String str) throws IOException
```

della `DataOutputStream` entrambe contenute nel package `java.io`.

I precedenti metodi permettono la lettura e scrittura dei caratteri di una stringa secondo il formato UTF-8.

Localizzazione: l'oggetto `java.util.Locale`

La classe `Locale` appartiene, come la classe `ResourceBundle`, al package `java.util` che contiene alcuni degli strumenti a supporto dell'internationalization. Il package `java.text` fornisce gli strumenti per la formattazione del testo, mentre in `java.util` sono stati aggiunti strumenti relativi alla localizzazione: infatti non solo è importante sapere come creare applicazioni portabili in funzione delle convenzioni geografiche ma è necessario anche poter sapere quale è il set di regole in uso, ovvero effettuare la localizzazione.

L'oggetto `Locale` serve proprio a questo, dato che, oltre a contenere le informazioni relative all'insieme di regole in uso entro un determinato ambito geografico/linguistico, permette di identificare l'ambito stesso.

In funzione dei parametri passati al costruttore è possibile creare un `Locale` funzione di una lingua, di una nazione o anche di alcuni parametri particolari.

Ad esempio il costruttore nella versione con il maggior numero di parametri ha la seguente firma

```
public Locale (String language, String country, String variant)
```

I valori dei parametri passati si riferiscono a precisi simboli definiti dallo standard ISO (International Standard Organization, l'organismo internazionale che cura a livello mondiale le regole di uniformazione).

Il primo di questi si riferisce al linguaggio ovvero alla lingua che ha predominanza nel particolare luogo, lingua che viene specificata tramite due lettere minuscole rappresentative della località e si riferisce allo standard ISO639. Per esempio la sigla per l'inglese in uso negli Stati Uniti è `en` mentre quella dell'italiano è `it`.

Va tenuto presente che non necessariamente in una stessa nazione si parla un'unica lingua. Per esempio esistono zone dell'Italia settentrionale in cui predominano il tedesco o il francese e ci sono nazioni come il Canada in cui, accanto a grandi aree di lingua inglese, esiste l'intero Quebec di lingua francese. Il secondo parametro del costruttore si riferisce proprio a questo: si tratta di una stringa costituita da caratteri questa volta maiuscoli (specificati nello standard ISO3166), che indicano la regione geografica: per la parte italiana della Svizzera ad esempio si utilizza la stringa `"CH"`.

Per la definizione di un oggetto `Locale` basterebbero queste due proprietà, ma esiste anche la possibilità di fornire una variante ovvero permettere la specifica di strutture personalizzate implementate, ad esempio, da particolari tipi di browser. Per rappresentare un oggetto `Locale` si utilizza spesso la notazione del tipo `language_country`.

Ovviamente affinché l'oggetto sia dotato di una certa utilità sono stati introdotti nel JDK 1.1 altri oggetti *locale sensitive*, il cui comportamento cioè sia dipendente dal `Locale` impostato. Ognuno di questi oggetti dispone di metodi che permettono di specificare il `Locale`, oppure che utilizzano quello di default (si vedranno degli esempi in seguito). I metodi più importanti dell'oggetto `Locale` sono quindi:

```
public static synchronized void setDefault();
public static synchronized Locale getDefault();
```

Si tratta di metodi di classe e permettono rispettivamente di settare il `Locale` di default oppure di vedere quale sia quello corrente. Per eseguire delle operazioni *locale sensitive* si avranno allora due possibilità:

- utilizzare il `Locale` di default senza specificarlo nei metodi *locale sensitive*;
- utilizzare la versione dei metodi che hanno l'oggetto `Locale` come parametro.

Ultima considerazione riguarda la presenza di `Locale` predefiniti relativi ai Paesi più importanti, messi a disposizione attraverso variabili statiche definite nella stessa classe `Locale`. Per ottenere quello italiano basterà scrivere

```
Locale italia = Locale.ITALY;
```

Creazione di risorse localizzate: oggetto `java.util.ResourceBundle`

Come si è avuto modo di accennare, un meccanismo che permetta una semplice commutazione da un linguaggio a un altro deve essere accompagnato da un modo per

conoscere la località di esecuzione: questo compito è svolto dall'oggetto `ResourceBundle`.

La filosofia di utilizzo di questo oggetto è molto simile a quella delle classi di supporto di un JavaBean. In questo caso insieme alla classe rappresentativa del componente, a supporto del componente possono esistere altre classi i cui nomi sono legati al nome della classe principale.

Per esempio, la classe che descrive esplicitamente il comportamento di un componente in termini di proprietà, eventi ed azioni, deve avere il nome `MiaClasseBeanInfo` dove `MiaClasse` è il nome della classe del Bean.

Si supponga di essere nella necessità di scrivere un componente che debba essere venduto in tutto il mondo e quindi debba adattarsi a ogni ambito geografico/linguistico. Se il nome di tale componente è `GlobalBean`, allora sue proprietà potranno essere descritte nella classe `GlobalBeanInfo`: in questo caso per permettere l'internazionalizzazione sarebbe sufficiente fornire una classe con tale nome per ogni località.

Così facendo si dovrebbe creare una classe per ogni località e quindi compilare più classi diverse. Se poi volessimo aggiungere una nuova località a quelle disponibili si dovrebbe programmare un'altra classe e compilarla perdendo in eleganza e flessibilità.

Inoltre il compito di scegliere quale classe `BeanInfo` utilizzare in funzione del locale dovrebbe essere in questo caso a totale carico del programmatore e comunque prevista in fase di progettazione.

Sicuramente l'alternativa più semplice ed elegante potrebbe essere quella di creare un'unica classe `GlobalBeanInfo` e fornire ad essa un meccanismo più semplice per visualizzare le proprietà del Bean in modo dipendente dalla località. Ad esempio si potrebbe pensare di creare un file di testo al quale la classe `BeanInfo` potrebbe accedere per reperire le informazioni di cui ha bisogno, ovvero i nomi di proprietà, eventi e metodi. Molto comodo sarebbe se la classe `GlobalBeanInfo` potesse accorgersi di questo file e ne utilizzasse autonomamente le informazioni.

La classe `ResourceBundle` permette di fornire le informazioni in modo *locale sensitive* attraverso gli stessi meccanismi con cui le classi `BeanInfo` le forniscono ai Bean, sia estendendo tale classe oppure utilizzando semplici file testo che contengono i valori da utilizzare.

Altra analogia fra i Bean e la internazionalizzazione è costituita dalla presenza di una regola per il nome che tali classi o file testo devono avere.

`ResourceBundle`: utilizzo e regole di naming

Per definire un insieme di risorse relative ad una particolare località, è possibile estendere la classe astratta `ResourceBundle` definendo i metodi:

```
protected abstract Object handleGetObject(String key) throws MissingResourceException

public abstract Enumeration getKeys()
```

Il primo ha come parametro un oggetto `String` che è il nome della risorsa cui si vuole accedere in modo *locale sensitive*. Il valore di ritorno sarà un `Object` rappresentativo della risorsa e adattato alla località. La sua assenza provoca una eccezione. Il metodo `getKeys()` invece ritorna l'insieme dei nomi delle risorse disponibili in quel `ResourceBundle`.

Tralasciando per il momento come questo sia possibile, si supponga di aver creato tanti oggetti `ResourceBundle` quante sono le località da gestire.

Primo passo da fare per accedere in modo *locale sensitive* a una risorsa è quella di ottenere l'oggetto `ResourceBundle` appropriato. Per fare questo si utilizzano le seguenti due versioni del metodo `static getBundle()`*:*

```
public static final ResourceBundle getBundle(String baseName)
   throws MissingResourceException

public static final ResourceBundle getBundle(String baseName, Locale locale)
```

Anche in questo caso, come in molti metodi relativi all'internationalization, sono disponibili due versioni dello stesso metodo che differiscono nella specificazione o meno dell'oggetto `Locale`.

Se si utilizza la prima versione viene considerato come oggetto `Locale` quello di default eventualmente precedentemente settato con il metodo `setDefault()`.

Si supponga adesso di disporre dell'oggetto `ResourceBundle` appropriato: per ottenere la risorsa relativa a una chiave particolare, basterà utilizzare il metodo `getObject()`, passando come parametro la chiave stessa: questo metodo ritornerà l'oggetto relativo alla chiave ed all'oggetto `Locale` utilizzato.

Al fine di evitare inutili operazioni di cast, dato che nella maggior parte dei casi gli oggetti sono delle stringhe, è fornito il metodo `getString()`.

Per quanto riguarda le regole di naming è bene tener presente che ogni risorsa ha un nome, il cosiddetto `baseName`, che deve essere esplicativo della risorsa che vogliamo gestire in modo *locale sensitive*.

Si supponga di voler gestire le etichette di due pulsanti in due lingue diverse, ad esempio italiano ed inglese, relativamente ai messaggi "SI" e "NO". Il nome di questa risorsa potrebbe essere `Bottone`.

In base alle regole di naming, il nome della classe che estende `ResourceBundle` deve essere quello della risorsa a cui sono aggiunte, nel modo descritto, le stringhe identificatrici della lingua.

Quindi la classe che contiene le label dei bottoni italiani dovrà avere nome `Bottone_it` e la sua implementazione potrebbe essere

```
public abstract class Bottone_it extends ResourceBundle {
   public Object handleGetObject(String key) {
      if (key.equals("yes")) return "SI";
```

```
      else if (key.equals("no")) return "NO";
         return null;
   } // fine handleObject
}
```

Analogamente la versione inglese si chiamerà Bottone_uk e potrebbe essere implementata nel seguente modo

```
public abstract class Bottone_uk extends ResourceBundle {
   public Object handleGetObject(String key) {
      if (key.equals("yes")) return "YES";
      else if (key.equals("no")) return "NO";
         return null;
   } // fine handleObject
}
```

Volendo creare un ResourceBundle per la parte italiana della Svizzera relativa a una minoranza indicata con il modificatore MI, il suo nome sarebbe stato Bottone_it_CH_MI.

Le classi create sono ancora astratte in quanto non è stato ridefinito il metodo getKeys() non utile al momento. Nel caso lo fosse, basterà creare un oggetto che implementi l'interfaccia java.util.Enumeration e ritornarlo come risultato.

Il fatto che le classi create sono astratte non è comunque un problema in quanto non vengono mai istanziate direttamente con un costruttore, ma sono ottenute attraverso il metodo getBundle().

Se l'oggetto Locale di default è Locale_ITALY, si otterrà l'oggetto ResourceBundle attraverso il metodo:

```
ResourceBundle rb = ResourceBundle.getBundle("bottoni");
```

Mentre per ottenere quello inglese si dovrà esplicitare il Locale differente e si utilizzerà la seguente forma

```
ResourceBundle rb = ResourceBundle.getBundle("bottoni", new Locale("uk"));
```

L'utilizzo dei file di testo

In alternativa all'utilizzo di ResourceBundle si può ricorrere all'utilizzo di semplice file di testo in cui introdurre le varie versioni internazionalizzate di una stessa risorsa: ovviamente questa eventualità risulta utilizzabile solamente nel caso in cui tali risorse siano stringhe.

Si supponga ad esempio di voler specificare un Locale nel seugente modo

```
Locale locale= new Locale("lingua", "paese", "modif");
```

Se la risorsa si chiama `bottoni`, alla esecuzione del metodo `getBundle()`, nello spazio del `ClassLoader` inizia la ricerca di una classe di nome `Bottoni_lingua_paese_modif`. Se questa non è presente viene allora cercato un file di estensione `properties` dal nome

```
Bottoni_lingua_paese_modif.properties
```

Questo file che segue lo standard dei file Properties, avrà le varie coppie chiave–valore elencate ciascuna su una riga divise da un segno di uguale (=).

Nel caso in cui nemmeno tale file venga trovato, si generalizza la ricerca a un livello più alto, andando prima a cercare la classe denominata `Bottoni_lingua_paese` e successivamente il file `Bottoni_lingua_paese.properties`.

In caso negativo si procede a una ulteriore generalizzazione ricercando la classe `Bottoni_lingua` e il file `Bottoni_lingua.properties`. Infine sono cercati la classe `Bottoni` e il file `Bottoni.properties`.

Se nessuna di queste è stata trovata viene sollevata una eccezione del tipo `MissingResourceException`. È da notare che nel caso in cui venisse trovata una delle classi o file cercati, la ricerca non sarà interrotta. Questo perché esiste la possibilità di estendere un `ResourceBundle` da un altro e, ad esempio, fare in modo che la risorsa `Bottoni_lingua_paese` erediti delle informazioni da `Bottoni_lingua`. Questa è una caratteristica da non sottovalutare.

Le classi `ListResourceBundle` e `PropertyResourceBundle`

Si è accennato all'esistenza di classi del tipo `ResourceBundle` già presenti nel JDK 1.1. Queste classi, sempre del package `java.util`, sono `ListResourceBundle` e `PropertyResourceBundle`.

La `ListResourceBundle` è una classe astratta che fornisce le informazioni sotto forma di vettore di coppie di oggetti.

Ciascuno di questi oggetti è un vettore di due elementi: il primo è una stringa indentificatrice del dato, il secondo è il dato stesso. La classe è astratta e il suo utilizzo prevede la sua estensione e la definizione del metodo:

```
protected abstract Object[][] getContents();
```

Nel caso dell'esempio precedente il `ListResourceBundle` associato alla versione italiana sarebbe stato

```
// Versione italiana della ListResourceBundle
public class Bottone_it extends ListResourceBundle {
   protected Object[][] getContents() {
      return {{"yes","SI"},{"no","NO"}};
```

```
   }
}
```

La `PropertyResourceBundle` permette di ottenere le informazioni a partire da un `InputStream`. Questa classe non è astratta e dispone del costruttore:

```
public PropertyResourceBundle(InputStream in);
```

In particolare verranno cercate, nello stream, le coppie del tipo *`chiave=valore`* e da esse saranno ottenute le informazioni. Nel caso dell'esempio visto prima si sarebbe dovuto creare un file di nome `inf.txt` dato da

```
# file contenente le informazioni usate dal
# PropertyResourceBundle italiano
yes = SI
no = NO
```

e si sarebbe dovuto utilizzare il seguente codice

```
try {
   FileInputStream fis = new FileInputStream(new File("inf.txt"));
   PropertyResourceBundle prb = new PropertyResourceBundle(fis);
   String yes_it = prb.getString("yes");
   ...
   fis.close
}
catch(IOException ioe) {
   ...
}
```

In questo esempio è stato volontariamente messo un nome qualsiasi per il file che contiene le informazioni in quanto, se si utilizza esplicitamente `PropertyResourceBundle`, non è necessario che siano soddisfatte le regole di naming sopra descritte.

Le due classi appena viste sono utilizzate in modo completamente trasparente nella fase di ricerca a seguito della esecuzione del metodo `getBundle()`.

A questo punto ci si potrebbe chiedere quando utilizzare un file di properties per specificare le grandezze *locale sensitive* e quando estendere la classe `ResourceBundle`.

Nel caso in cui le modifiche da fare siano relative solamente a del testo è bene utilizzare i file Properties che permettono di creare facilmente le varie versioni con un semplice text editor. Nel caso in cui le modifiche siano relative a oggetti non vi è altra scelta che creare le varie classi `ResourceBundle` ricordandosi, però, della possibilità di estendere una dall'altra e di ereditarne le informazioni.

La formattazione

Un aspetto molto importante legato all'internazionalizzazione di una applicazione è dato dalla gestione dei diversi formati di visualizzazione di stringhe particolari, come date, valute o altro.

La gestione delle date è forse uno dei problemi più frequenti quando ad esempio si devono manipolare dati memorizzati in tabelle di database, a causa dei differenti formati utilizzati. Il fenomeno legato all'anno 2000 ha insegnato molto da questo punto di vista.

Se relativamente alla gestione delle varie etichette della interfaccia grafica, la localizzazione permette di risolvere alcuni aspetti prettamente estetici, in questo caso l'utilizzo delle tecniche di internazionalizzazione permette di evitare l'insorgere di errori di funzionamento.

Per questo motivo la corretta formattazione può essere considerata un aspetto ancora più importante rispetto alla gestione delle risorse in modalità *locale sensitive*. Il primo aspetto che si può prendere in considerazione è dato dalla formattazione dei numeri

La formattazione dei numeri

La classe `NumberFormat` è molto utile in tal senso dato che permette di formattare numeri, valute e percentuali.

Nel caso dei numeri è possibile formattare sia valori di tipo primitivo come `double` o `int` sia variabili reference, come `Double` o `Integer`.

Il metodo `getNumberInstance()` permette di ottenere un formattatore specifico in funzione del `Locale` impostato.

Ad esempio si potrebbe scrivere un piccolo programma `NumberFormatter` che formatta un numero in funzione del `Locale` specificato:

```
// due formattatori specifici
NumberFormat ItalianFormatter, EnglishFormatter;

// creazione dei Locale italiano e inglese americano
Locale ItalianLocale = new Locale("it", "IT");
Locale USLocale = new Locale("en", "US");

String EnglishTotalAmount, ItalianTotalAmount;

// variabile da formattare
Double TotalAmount = new Double (3425759.456);

// crea il formattatore italiano e formatta
ItalianFormatter = NumberFormat.getNumberInstance(ItalianLocale);
ItalianTotalAmount = ItalianFormatter.format(TotalAmount);
System.out.println("Formattazione " + Locale.ITALIAN + " :" + ItalianTotalAmount);

// crea il formattatore inglese americano e formatta
```

```
EnglishFormatter = NumberFormat.getNumberInstance(USLocale);
EnglishTotalAmount = EnglishFormatter.format(TotalAmount);
System.out.println("Formattazione " + Locale.ENGLISH + " :" + EnglishTotalAmount);
```

Tale programma produce il seguente risultato

```
Formattazione it : 3.425.759,456
Formattazione en : 3,425,759.456
```

La formattazione di valute in funzione del locale scelto avviene in maniera del tutto analoga, con la sola differenza che il metodo invocato per ottenere un formattatore è in queso caso il `getCurrencyInstance()`. Ad esempio, nel programma `CurrencyFormatter` si trova il seguente codice

```
Double TotalAmount = new Double (3425759.456);

NumberFormat ItalianFormatter, EnglishFormatter;

String EnglishTotalAmount, ItalianTotalAmount;

Locale ItalianLocale = new Locale("it", "IT");
ItalianFormatter = NumberFormat.getCurrencyInstance(ItalianLocale);
ItalianTotalAmount = ItalianFormatter.format(TotalAmount);
System.out.println("Somma in Lire: " + ItalianTotalAmount);

Locale USLocale = new Locale("en", "US");
EnglishFormatter = NumberFormat.getCurrencyInstance(USLocale);
EnglishTotalAmount = EnglishFormatter.format(TotalAmount);
System.out.println("Somma in Dollari: " + EnglishTotalAmount);
```

Il risultato dell'esecuzione in questo caso sarebbe

```
Somma in Lire: L. 3.425.759
Somma in Dollari: $3,425,759.46
```

Ovviamente la trasformazione in funzione del `Locale` è valida solo da un punto di vista matematico, e non dall'effettivo cambio monetario delle valute utilizzate.

Meno importante in questo caso l'utilizzo di `Locale` differenti, anche se di fatto essi permettono di realizzare applicazioni più flessibili.

Formattazione personalizzata di cifre

Nel caso di valori corrispondenti a cifre decimali è possibile utilizzare le classi `DecimalFormat` e `DecimalFormatSymbols` sia per effettuare una formattazione

personalizzata dei valori, indicando ad esempio il numero di zeri prefissi e postfissi, il carattere separatore delle migliaia e dei decimali.

In questo caso, al fine di personalizzare la formattazione di un valore decimale, la prima cosa da fare è definire il pattern di formattazione utilizzando le seguenti regole in formato BNF

```
pattern     := subpattern{;subpattern}
subpattern  := {prefix}integer{.fraction}{suffix}
prefix      := '\\u0000'..'\\uFFFD' - specialCharacters
suffix      := '\\u0000'..'\\uFFFD' - specialCharacters
integer     := '#'* '0'* '0'
fraction    := '0'* '#'*
```

dove le notazioni hanno il significato riportato nella seguente tabella.

Tabella 12.2

notazione	descrizione
`X*`	0 o più istanze di X
`(X | Y)`	X o Y
`X..Y`	un carattere qualsiasi da X a Y inclusi
`S - T`	tutti i caratteri in S eccetto quelli in T
`{X}`	X è opzionale

Nello schema precedente il primo sottopattern è relativo ai numeri positivi, il secondo a quelli negativi. In ogni sottopattern è possibile specificare caratteri speciali che sono indicati nella tab. 12.3.

Alcuni esempi di pattern utilizzati di frequente sono riportati nella tab. 12.4.

Un esempio potrebbe essere il seguente:

```
Locale ItalianLocale = new Locale("it", "IT");
NumberFormat NumberFormatter = NumberFormat.getNumberInstance(ItalianLocale);
DecimalFormat DecimalFormatter = (DecimalFormat) NumberFormatter;
DecimalFormatter.applyPattern(pattern);
```

Tabella 12.3

notazione	descrizione
0	una cifra
#	una cifra, zero equivale ad assente
.	separatore decimale
,	separatore per gruppi di cifre
E	separa mantissa ed esponente nei formati esponenziali
;	separa i formati
-	prefisso per i negativi
%	moltiplica per 100 e mostra come percentuale
?	moltiplica per 1000 e mostra come per mille
{	segno di valuta; sostituito dal simbolo di valuta; se raddoppiato, sostituito dal simbolo internazionale di valuta; se presente in un pattern, il separatore decimale monetario viene usato al posto del separatore decimale
X	qualsiasi altro carattere può essere usato nel prefisso o nel suffisso
'	usato per indicare caratteri speciali in un prefisso o in un suffisso

Tabella 12.4

Output del programma `DecimalFormatDemo`			
value	pattern	output	spiegazione
123456.789	###,###.###	123,456.789	Il segno del "diesis" (#) rappresenta una cifra, la virgola (,) è il separatore di gruppi di cifre e il punto (.) è il separatore decimale.
123456.789	###.##	123456.79	Il value ha tre cifre a destra del punto decimale ma il pattern ne presenta solo due. Il metodo `format` risolve questa situazione con l'arrotondamento a due cifre.
123.78	000000.000	000123.780	Il pattern specifica degli zero prima e dopo i numeri significativi poiché, invece del diesis (#), viene usato il carattere 0.
12345.67	$###,###.###	$12,345.67	Il primo carattere nel pattern è il segno del dollaro ($). Da notare che nell'output formattato esso precede immediatamente la cifra più a sinistra.
12345.67	\u00A5###,###.###	¥12,345.67	Il pattern specifica il segno di valuta per lo yen giapponese (¥) con il valore Unicode 00A5.

```
String FormattedValue = DecimalFormatter.format(value);
System.out.println(pattern + " " + FormattedValue + " " + loc.toString());
```

che in funzione del pattern utilizzato e del locale impostato produce i risultati riportati in tab. 12.5.

Tabella 12.5

pattern	locale	risultato
###,###.###	en_US	123,456.789
###,###.###	de_DE	123.456,789
### ###.###	fr_FR	123 456,789

Formattazione personalizzata

Grazie all'utilizzo della classe `DecimalFormatSymbols` è possibile modificare i vari simboli utilizzati come separatori: ad esempio è possibile specificare il carattere di separazione delle migliaia, dei decimali, il segno meno, quello di percentuale e quello di infinito.

Ad esempio

```
DecimalFormatSymbols MySymbols = new DecimalFormatSymbols(currentLocale);
unusualSymbols.setDecimalSeparator('-');
unusualSymbols.setGroupingSeparator('@');

String MyPattern = "#,##0.###";
DecimalFormat weirdFormatter = new DecimalFormat(MyPattern, MySymbols);
weirdFormatter.setGroupingSize(4);

String bizarre = weirdFormatter.format(235412.742);
System.out.println(bizarre);
```

L'esecuzione di questa porzione di codice produce il seguente risultato

```
2@35412-742
```

Formattazione di date e orari

Uno degli aspetti più importanti legati alla rappresentazioni di dati indipendentemente dalla zona geografica è dato dalla corretta gestione delle date e degli orari.

La classe `DateFormat` da questo punto di vista rappresenta un valido ausilio per tutte le operazioni di rappresentazione *locale independent* e di formattazione.

Al solito è possibile utilizzare formati standard oppure effettuare formattazioni sulla base di pattern particolari. Come prima cosa si prende in esame il caso della formattazione basata su formati predefiniti.

Formattazione predefinita

Per effettuare la formattazione di una data si deve per prima cosa creare un formattatore tramite il metodo `getDateInstance()` della classe `DateFormat`.

Successivamente, in modo del tutto analogo ai casi precedenti è necessario invocare il metodo `format()` per ottenere una rappresentazione testuale della data secondo il `Locale` utilizzato.

Ad esempio la parte centrale del programma `DateFormatter` potrebbe essere

```
// Locale italiano
Locale ItalianLocale = new Locale("it", "IT");
int Style = DateFormat.DEFAULT;
DateFormat ItalianDateFormatter;
ItalianDateFormatter = DateFormat.getDateInstance(Style, ItalianLocale);
String ItalianDate = ItalianDateFormatter.format(today);
System.out.println("Data formattata secondo lo standard italiano: " + ItalianDate);
```

Variando il `Locale` utilizzato, si otterrebbe il seguente risultato

```
Data formattata secondo lo standard italiano: 9-ago-00
Data formattata secondo lo standard USA: Aug 9, 2000
Data formattata secondo lo standard tedesco: 09.08.2000
```

Il metodo `getDateInstance()` viene fornito in tre versioni differenti

```
public static final DateFormat getDateInstance()
public static final DateFormat getDateInstance(int style)
public static final DateFormat getDateInstance(int style, Locale locale)
```

Mentre il parametro `locale` se specificato permette di ottenere un formattatore specializzato, il parametro `style` serve per specificare lo stile di visualizzazione della data: ad esempio, utilizzando il locale italiano e variando fra i valori `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` e `FULL`, si possono ottenere i seguenti risultati

```
DEFAULT: 9-ago-00
SHORT:      09/08/00
MEDIUM:     9-ago-00
```

```
LONG:    9 agosto 2000
FULL:    mercoledì 9 agosto 2000
```

La versione senza parametri di `getDateInstance()` restituisce un formattatore in base al `Locale` correntemente installato e con stile `DEFAULT`.

Nel caso invece che si desideri formattare orari si dovrà ricavare un formattatore opportuno per mezzo del metodo `getDateTimeInstance()` disponibile anche in questo caso in quattro versioni differenti a seconda che si voglia specificare il `Locale` e lo stile

```
public static final DateFormat getDateTimeInstance()
public static final DateFormat getDateTimeInstance(int dateStyle)
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle,
                                                    Locale aLocale)
```

Rispetto a prima, adesso il programma `TimeFormatter` avrebbe di diverso solamente la parte di definzione del formattatore

```
DateFormat ItalianTimeFormatter;
ItalianTimeFormatter = DateFormat.getTimeInstance(DateFormat.DEFAULT, ItalianLocale);
```

Anche in questo caso la scelta dello stile di rappresentazione influisce sul risultato finale: variando fra `DEFAULT` e `FULL` si potranno ottenere i seguenti risultati

```
DEFAULT: 11.48.04
SHORT:      11.48
MEDIUM:     11.48.04
LONG:    11.48.04 GMT-07:00
FULL:    11.48.04 GMT-07:00
```

Se infine si desidera effettuare formattazioni di date e orari contemporaneamente si potrà ottenere il formattatore adatto grazie al metodo

```
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle,
                                                    Locale locale)
```

Il risultato che si ottiene dopo l'esecuzione utilizzando il `Locale` italiano e variando stile per la data e per l'orario è il seguente

```
DEFAULT: 9-ago-00 12.23.22
```

```
SHORT:      09/08/00 12.23
MEDIUM:     9-ago-00 12.23.22
LONG:       9 agosto 2000 12.23.22 GMT-07:00
FULL:       mercoledì 9 agosto 2000 12.23.22 GMT-07:00
```

Formattazione personalizzata delle date

Facendo utilizzo della classe `SimpleDataFormatter` si può definire un pattern di formattazione e personalizzare il formato voluto. Ecco di seguito un breve esempio

```
// Crea un oggetto con la data attuale
// da cui partire per la formattazione
Date now = new Date();

// prepara un formattatore per formattare una data
// nella forma 19/08/2000 12.09.59
String pattern = "dd/MM/yyyy H.mm.ss";
Locale ItLocale = new Locale("it","IT");
SimpleDateFormat DateTimeFormatter = new SimpleDateFormat(pattern, ItLocale);
String now_formatted = DateTimeFormatter.format(now);
```

Il funzionamento di questa classe è piuttosto intuitivo; per una completa rassegna dei caratteri utilizzabili e del loro significato per la composizione della stringa `pattern` si può far riferimento alla tab. 12.6 riportata nella pagina successiva.

Ogni carattere ad eccezione di quelli contenuti in `['a'..'z']` e `['A'..'Z']` viene utilizzato come separatore. Ad esempio i caratteri

`:` `.` [*spazio*] `,` `#` `@`

appariranno anche se non racchiusi fra apici.

Ad esempio i seguenti pattern producono i risultati riportati di seguito

```
"yyyy.MM.dd G 'at' hh:mm:ss z"   ->     1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, ''yy"               ->     Wed, July 10, '96
"h:mm a"                         ->     12:08 PM
"hh 'o''clock' a, zzzz"          ->     12 o'clock PM, Pacific Daylight Time
"K:mm a, z"                      ->     0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"   ->     1996.July.10 AD 12:08 PM
```

Il programma contenuto nella classe `SimpleDateFormat` contiene inoltre un interessante e utile metodo, `parse()`, il quale è in grado di eseguire la traduzione inversa da una stringa rappresentante una data a oggetto `Date`.

Tabella 12.6

simbolo	significato	risultato	esempio
`G`	era	testo	`AD`
`y`	anno	numerico	`1996`
`M`	mese dell'anno	testo & numerico	`luglio & 07`
`d`	giorno del mese	numerico	`10`
`h`	ora in am/pm (1~12)	numerico	`12`
`H`	ora del giorno(0~23)	numerico	`0`
`m`	minuto dell'ora	numerico	`30`
`s`	secondi del minuto	numerico	`55`
`S`	millisecondi	numerico	`978`
`E`	giorno della settimana	testo	`giovedì`
`D`	giorno dell'anno	numerico	`189`
`F`	giorno della settimana	numerico	`2 (2° mer di luglio)`
`w`	settimana dell'anno	numerico	`27`
`W`	settimana del mese	numerico	`2`
`a`	marcatore am/pm	testo	`PM`
`k`	ora del giorno (1~24)	numerico	`24`
`K`	ora in am/pm (0~11)	numerico	`0`
`z`	timezone	testo	`Pacific Standard Time`
`'`	escape per testo	delimitatore	
`''`	apice singolo	carattere	`'`

In questo caso si tenga presente che la mancanza di informazione derivante dall'utilizzo di sole due cifre per l'anno viene risolta in maniera drastica: viene infatti effettuato un aggiustamento della data considerando secolo corrente quello compreso dagli 80 anni prima ai 20 dopo la data di creazione dell'istanza della `SimpleDateFormat`. In base a tale regola, utilizzando un pattern MM/dd/yy ed un `SimpleDateFormat` creato il 12 gennaio 1999, gli estremi per considerare il secolo saranno 1919 e 2018. Quindi il numero 64 verrà considerato come 1964, mentre il 17 come 2017, come riportato in fig. 12.2.

Gestione dei messaggi

Si è visto finora come modificare contenuto o formattazione di una stringa in funzione del `Locale` impostato. Anche se la flessibilità ottenibile con tali tecniche è sicuramente alta, si tratta pur sempre di utilizzare archivi di messaggi prestabiliti e non modificabili.

Si consideri ad esempio la gestione di messaggi di errore che devono essere visualizzati in apposite finestre di dialogo. In questo caso particolare infatti il testo che deve essere visualizzato spesso non può essere prestabilito a priori, essendo dipendente dal caso specifico.

Supponendo di dover visualizzare un messaggio che indichi il numero di messaggi di posta elettronica presenti in una determinata casella, si potrebbe pensare di preparare i seguenti messaggi

```
"Nella casella " + PopAccount + " ci sono " + NumMessages + " messaggi"
```

oppure

```
"Nella casella" + PopAccount + " non ci sono messaggi"
```

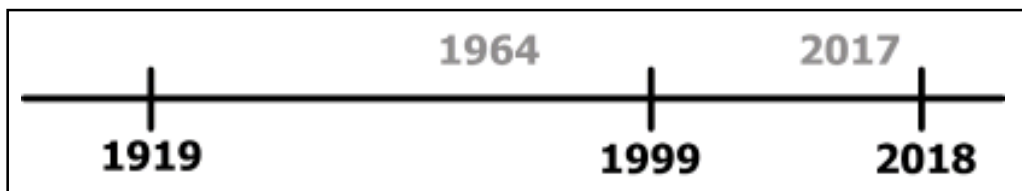
Queste due semplici opzioni non garantiscono sufficiente flessibilità, se si pensa al caso in cui sia presente un solo messaggio sul server, caso in cui si dovrebbe ricorrere ad uno schema del tipo

```
"Nella casella " + PopAccount + " vi è un solo messaggio"
```

Inoltre non tutte le lingue utilizzano lo stesso schema soggetto–verbo–complemento per cui a volte non è sufficiente rimpiazzare le parole che cambiano al variare della lingua, ma si deve modificare tutta la struttura della frase.

Una soluzione potrebbe essere quella di aggiungere un numero sufficiente di template in modo da coprire tutte le possibili casistiche: questa strada, oltre a non essere particolarmente elegante, complica le cose nel caso in cui, oltre alla gestione del singolare/plurale, si deve tener conto anche del genere (maschile/femminile).

In definitiva la gestione di messaggi di vario tipo deve tener conto della gestione delle frasi composte e del plurale.

Figura 12.2 – *Nel caso in cui l’anno di partenza sia espresso con due sole cifre, il metodo parse effettua un’approssimazione*

Messaggi composti

Riprendendo in esame il caso precedente, si immagini di voler gestire una frase del tipo

```
Alle ore 12.23 del giorno 09/08/00 nella casella x34f erano presenti 33 messaggi.
```

dove le parole sottolineate rappresentano le parti variabili della frase. Per il momento si tralascia il problema del singolare della parola "messaggi".

Successivamente all'individuazione delle parti variabili del messaggio si deve creare un pattern che corrisponda alla frase da gestire, pattern che potrà essere memorizzato in un file di proprietà; ad esempio tale file potrebbe essere così strutturato

```
template = Alle ore {2,time,short} del giorno
{2,date, long} nella casella {0} erano presenti {1,number,integer} messaggi.
mailbox = x34f
```

la prima riga di tale file contiene il pattern generico da utilizzare per ottenere la frase finale. Le parentesi graffe contengono le parti variabili che dovranno essere sostituite di volta in volta, e il loro significato è riportato nella tab. 12.7.

È necessario creare quindi un array di oggetti dove memorizzare le varie parti da sostituire nel pattern.

Tabella 12.7

argomenti per `template` in `MessageBundle_en_US.properties`	
argomento	**descrizione**
`{2,time,short}`	La porzione relativa al tempo dell'oggetto `Date`. Lo stile `short` specifica lo stile di formattazione `DateFormat.SHORT`.
`{2,date,long}`	La porzione relativa alla data dell'oggetto `Date`. Lo stesso oggetto `Date` viene usato per le variabili sia di data che di tempo. Nell'array di argomenti di `Object`, l'indice dell'elemento che detiene l'oggetto `Date` è `2`. (Vedere successiva descrizione).
`{1,number,integer}`	Un oggetto `Number` ulteriormente specificato con lo stile numerico integer.
`{0}`	La `String` nel `ResourceBundle` corrispondente alla chiave `planet`.

```
Object[] messageArguments = {
   messages.getString("planet"),
   new Integer(7),
   new Date()
};
```

Ovviamente si potrebbe pensare di rendere le cose ancora più eleganti e automatizzate creando un file di risorse anche per queste variabili.

Il passo successivo consiste nel creare un formattatore di messaggi, ovvero

```
MessageFormat formatter = new MessageFormat("");
formatter.setLocale(ItalianLocale);
```

sul quale si applica il `Locale` corrente per permettere la corretta visualizzazione di date e simili.

Infine si deve provvedere ad applicare il formattatore al pattern, sostituendo i parametri variabili con quelli presenti nell'array di oggetti appena visto:

```
formatter.applyPattern(MailMessages.getString("template"));
String msg = formatter.format(MessagesArguments);
```

Il risultato che si ottiene nel caso in cui il locale sia quello italiano

```
Alle ore 13.42 del giorno 10 agosto 2000 nella casella
X34f erano presenti 33 messaggi.
```

Mentre per quello inglese statunitense

```
At 1:41 PM on August 10, 2000 the message box X34f contains 33 messages.
```

Confronto fra caratteri

Molto di frequente può accadere di dover controllare se un certo carattere corrisponda a una cifra oppure a una lettera: si pensi ad esempio a tutti i controlli che spesso sono necessari sui dati immessi dall'utente tramite un form. In questo caso la soluzione che tipicamente si adotta è quella di confrontare il valore del codice (ASCII o più propriamente Unicode) dei caratteri immessi: ad esempio si potrebbe scrivere

```
char c;

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
   // c  è una lettera

if (c >= '0' && c <= '9')
```

```
   // c  è un numero

if ((c == ' ') || (c == '\n') || (c == '\t'))
   // c  è una carattere speciale
```

Benché questa sia la soluzione tipicamente utilizzata nella maggior parte dei casi, è errata, dato che effettua un semplice controllo sull'ordinamento fornito dai codici Unicode dei caratteri, ma non prende in considerazione le regole grammaticali presenti nei vari linguaggi.

Per ovviare a questo problema si possono utilizzare i vari metodi messi a disposizione dalla classe `Character`, quali

```
public static boolean isDigit(char ch)
public static boolean isLetter(char ch)
public static boolean isLetterOrDigit(char ch)
public static boolean isLowerCase(char ch)
public static boolean isUpperCase(char ch)
```

Utilizzando tali metodi il controllo sui caratteri può essere effettuato ad esempio nel seguente modo

```
char c;
if (Character.isLetter(ch))
   ...
if (Character.isDigit(ch))
   ...
if (Character.isSpaceChar(ch))
   ...
```

Inoltre il metodo `getType()` permette di ricavare la categoria di un carattere: nel caso di caratteri dell'alfabeto è possibile ricavare se si tratta di una lettera maiuscola o minuscola, oppure nel caso di caratteri speciali se si tratta di un simbolo matematico o di un simbolo di punteggiatura.

Per maggiori approfondimenti su questo genere di indicazioni si consulti la documentazione ufficiale della Sun.

Ordinamento

Oltre a voler conoscere la categoria di un carattere, un altro tipo di operazione molto utile è determinare l'ordinamento dei caratteri, al fine ad esempio di elencare in ordine alfabetico una serie di parole. Per ottenere risultati corretti, anche in questo caso è indispensabile tenere conto delle regole grammaticali della lingua e zona geografica scelte.

Infatti, anche se può apparire ovvio che il carattere “a” sia sempre minore del carattere “b”, si possono avere situazioni meno ovvie, come il caso delle lettere accentate o di lettere doppie. Inoltre in alcuni rari casi potrebbe essere necessario specificare un nuovo set di regole di precedenza, senza dover per forza rispettare quelle del `Locale` in uso. La classe `Collator` a tal proposito permette di effettuare gli ordinamenti sulla base delle regole grammaticali della lingua utilizzata.

Per ottenere un `Collator` si può ad esempio scrivere

```
Collator DefaultCollator = Collator.getInstance();
```

se si vuole utilizzare la lingua del `Locale` di default, oppure

```
Collator ItalianCollator = Collator.getInstance(Locale.ITALIAN);
```

nel caso si voglia specificare il `Locale` da utilizzare.

Come si può notare in questo caso è sufficiente specificare la lingua e non anche il Paese, dato che si assume che le regole di ordinamento non dipendono dalla localizzazione geografica, ma esclusivamente dalla grammatica.

Una volta in possesso di un `Collator` non è necessario indicare nessun altro parametro, dato che le regole grammaticali sono contenute all'interno della classe, e si potrà quindi utilizzare il metodo compare per effettuare le comparazioni fra caratteri stringhe o semplicemente fra caratteri. Ad esempio

```
ItalianCollator.compare("aaa", "aab")
```

restituirà `-1` visto che nella nostra lingua la stringa `"aaa"` è minore rispetto ad `"aab"`.

Utilizzando due `Collator` differenti, uno per l'Italiano e l'altro per il Francese, si può provare a effettuare un ordinamento sul set di parole

```
{perché, péché, è, pêche, pesce}
```

Nel primo caso si ottiene

```
è, péché, pêche, perché, pesce
```

Mentre per la lingua francese il risultato è

```
è, pêche, péché, perché, pesce
```

Nel caso in cui si desideri utilizzare un set di regole di ordinamento personalizzate si può ricorrere alla classe `RuleBasedCollator`, specificando le regole di ordinamento per mezzo di una ótringa che viene poi passata al costruttore `RuleBasedCollator`. Ad esempio si potrebbe scrivere

```
String MyRule = "< a < b < c < d";
RuleBasedCollator MyCollator = new RuleBasedCollator(MyRule);
```

In questo caso il `Collator` creato effettuare le comparazioni utilizzando come unica regola base il fatto che la lettera “a” è minore di “b” che a sua volta è minore di “c” e di “d”.

Ovviamente affinché abbia senso utilizzare un set di regole personalizzate, è necessario specificare un ordinamento completo su tutte le lettere dell’alfabeto e simboli vari che si suppone saranno utilizzate nel programma. Prendendo ad esempio il caso riportato nella documentazione della Sun, si potrebbe pensare di creare due set di regole di ordinamento, uno per la lingua inglese e uno per la lingua spagnola.

```
// ordinamento lingua inglese
String enRules = ("< a,A < b,B < c,C < d,D < e,E < f,F "
                  + "< g,G < h,H < i,I < j,J < k,K < l,L "
                  + "< m,M < n,N < o,O < p,P < q,Q < r,R "
                  + "< s,S < t,T < u,U < v,V < w,W < x,X "
                  + "< y,Y < z,Z");

// ordinamento lingua spagnola
String smallnTilde = new String("\u00F1"); // ñ
String capitalNTilde = new String("\u00D1"); // Ñ

String spRules = ("< a,A < b,B < c,C "
                  + "< ch, cH, Ch, CH "
                  + "< d,D < e,E < f,F "
                  + "< g,G < h,H < i,I < j,J < k,K < l,L "
                  + "< ll, lL, Ll, LL "
                  + "< m,M < n,N "
                  + "< " + smallnTilde + "," + capitalNTilde + " "
                  + "< o,O < p,P < q,Q < r,R "
                  + "< s,S < t,T < u,U < v,V < w,W < x,X "
                  + "< y,Y < z,Z");
```

Si noti come in questo caso siano stati introdotti i caratteri speciali “ñ” e “Ñ” e come la doppia “ll” sia stata forzatamente inserita dopo la singola “l”, cosa che rende vera la relazione

```
"lu" < "ll"
```

Il programma principale potrebbe quindi effettuare l’ordinamento utilizzando i due `Locale` inglese e spagnolo, come ad esempio

```
try {
   RuleBasedCollator enCollator = new RuleBasedCollator(enRules);
   RuleBasedCollator spCollator = new RuleBasedCollator(spRules);
```

```
   sortStrings(enCollator, words);
   printStrings(words);

   System.out.println();

   sortStrings(spCollator, words);
   printStrings(words);
}
catch (ParseException pe) {
   System.out.println("Parse exception for rules");
}
```

dove `words` è un array di stringhe.

Migliorare l'efficienza delle operazioni di ordinamento

L'operazione di comparazione può risultare costosa da un punto di vista computazionale, per cui, quando si debbano svolgere frequentemente ordinamenti sullo stesso insieme di parole, può essere conveniente utilizzare una tecnica alternativa, basata sull'oggetto `CollationKey`.

In questo caso si genera una chiave per ogni parola da confrontare, e si provvede a effettuare successivamente la comparazione sulle chiavi, piuttosto che direttamente sulle parole.

Dato che la generazione delle chiavi è una operazione costosa, risulta conveniente utilizzare questa tecnica solo nel caso in cui si debba effettuare più volte la comparazione sullo stesso insieme di parole, in modo da ammortizzare il tempo iniziale necessario per la produzione delle chiavi.

Qui di seguito sono riportati alcuni passi del programma completo di ordinamento basato su chiavi:

```
Locale ItalianLocale = new Locale ("it", IT");
Collator ItalianCollator = Collator.getInstance (ItalianLocale);
CollationKey[] keys = new CollationKey[words.length];

for (int k = 0; k < keys.length; k ++)
   keys[k] = ItalianCollator.getCollationKey(words[k]);

// ordina gli elementi dell'array delle chiavi
CollationKey tmp;
for (int i = 0; i < keys.length; i++) {
   for (int j = i + 1; j < keys.length; j++) {
      if (keys[i].compareTo(keys[j]) > 0) {
         tmp = keys[i];
         keys[i] = keys[j];
         keys[j] = tmp;
```

```
        }
    }
}
```

dove al solito `words` è un array di stringhe. Una volta che l'array delle chiavi è ordinato, si può risalire alle stringhe originarie per mezzo del metodo `getResourceString()`, come ad esempio

```
for (int i = 0; i < keys.length; i++) {
    System.out.println(keys[i].getSourceString());
}
```

La classe `BreakIterator`

Dato un testo espresso in una determinata lingua, il processo di separazione in caratteri, parole, linee o frasi è sicuramente dipendente dalle regole grammaticali in vigore in tale lingua.

Va considerato che in certe lingue alcune parole sono espresse da un solo carattere del particolare alfabeto, in altre lingue si scrive da destra verso sinistra, in alcune si trascrivono le sole consonanti.

Ad esempio, la parola araba طريق, "ṭarīq", che significa "strada", può essere rappresentata dai caratteri Unicode

```
String street = "\ufed6" + "\ufef3" + "\u0631" + "\ufec1";
```

Questo semplice esempio fa comprendere come in realtà l'individuazione dei caratteri non sia un compito semplice.

La classe `BreakIterator`, per mezzo dei metodi

```
public static BreakIterator getCharacterInstance()
public static BreakIterator getCharacterInstance(Locale where)
public static BreakIterator getWordInstance()
public static BreakIterator getWordInstance(Locale where)
public static BreakIterator getSentenceInstance()
public static BreakIterator getSentenceInstance(Locale where)
public static BreakIterator getLineInstance()
public static BreakIterator getLineInstance(Locale where)
```

permette di ottenere delle istanze specifiche in funzione del tipo di separatore che si intende ottenere.

Il tipo di `BreakIterator` più semplice è quello che permette di individuare i singoli caratteri all'interno di una stringa. Eccone un semplice esempio

```
Locale ItLocale = new Locale ("it", IT");
BreakIterator ItalianCharIterator;
ItCharIterator  = BreakIterator.getCharacterInstance(ItLocale);
ItCharIterator.setText(StringToBreak);
int CharPosition = ItCharIterator.first();

while (CharPosition != BreakIterator.DONE) {
   System.out.println (CharPosition);
      CharPosition = ItCharIterator.next();
}
```

dove `StringToBreak` rappresenta una stringa da suddividere in caratteri, e che, nel caso della parola araba di cui prima, restituirebbe la sequenza `0,2,4,6`.

L'utilizzo degli altri metodi è similare e molto intuitivo, per cui si rimanda agli esempi allegati per maggiori approfondimenti o alla documentazione ufficiale della Sun.

Java e XML

DI ANDREA GIOVANNINI

Introduzione

XML è una tecnologia avviata a un utilizzo pervasivo nel Web Publishing e nelle applicazioni business to business. Java è il linguaggio ideale per questo tipo di applicazioni, grazie alle sue caratteristiche di portabilità ed estendibilità, ed è quindi naturale che le due tecnologie si incontrino. In questo capitolo si vedranno i concetti principali di XML e si approfondiranno alcuni temi specifici della programmazione di applicazioni XML con Java.

Fondamenti di XML

Si affronteranno ora gli aspetti principali di questa tecnologia, riprendendo poi con qualche esempio di programmazione Java i concetti presentati. Questa sezione non vuole comunque essere un tutorial completo di XML ma una semplice introduzione agli aspetti più avanzati della tecnologia che verranno affrontati nel prosieguo del capitolo.

Che cosa è XML

XML (eXtensible Markup Language) [1] è un metalinguaggio per definire nuovi linguaggi basati su tag. Per questo aspetto XML è molto simile a HTML ma c'è una differenza fondamentale: XML è una notazione per definire linguaggi mentre HTML è un particolare linguaggio. I punti chiave di XML sono i seguenti:

- Separazione del contenuto dalla rappresentazione dei dati: un documento XML definisce la struttura dei dati e non contiene alcun dettaglio relativo alla loro formattazione o a un qualsiasi utilizzo.

- Definizione di un formato standard: XML è uno standard del W3C e quindi un documento XML può essere elaborato da qualsiasi parser o tool conforme allo standard.

Segue ora un semplice esempio di file XML:

```
<?xml version="1.0"?>
<todolist>
   <item>
      <number>1</number>
      <priority>6</priority>
      <description>Leggere la posta</description>
      <state>2</state>
   </item>
   <item>
      <number>2</number>
      <priority>9</priority>
      <description>Riunione</description>
      <state>2</state>
   </item>
   <item>
      <number>3</number>
      <priority>8</priority>
      <description>Andare a correre nel parco</description>
      <state>1</state>
   </item>
</todolist>
```

Il documento precedente definisce una *todo-list*, ovvero un elenco di attività e di informazioni ad esse associate come la priorità e lo stato di avanzamento. Il primo elemento del documento è l'intestazione tipica di ogni file XML

```
<?xml version="1.0"?>
```

Quindi il documento presenta le informazioni relative alla todo-list organizzate in una gerarchia di elementi; i tag utilizzati sono funzionali alla struttura del documento e non contengono alcuna informazione relativa all'utilizzo della todo-list. Ad esempio dal documento è chiaro che l'attività `Riunione` ha una priorità maggiore rispetto a `Leggere la posta` ma non è possibile evincere nulla relativamente a come la todo-list possa essere visualizzata e in che tipo di documento.

Struttura di un documento XML

Così come un documento HTML deve essere redatto secondo alcune regole sintattiche ben precise, affinché sia visualizzabile da un browser, anche un documento XML deve

rispettare ben precise regole strutturali; in particolare

- ogni tag aperto deve essere chiuso;
- i tag non devono essere sovrapposti;
- i valori degli attributi devono essere racchiusi fra `" "`;
- i caratteri `<` `>` `"` `"` nel testo di un file XML devono essere rappresentati dai caratteri speciali `<` `>` e `"`

I documenti XML che rispettano le regole precedenti vengono detti ben formati (*well-formed*). Oltre a poter definire documenti ben formati è possibile specificare la particolare struttura di un file XML ad esempio per garantire che il tag description sia compreso all'interno del tag item. A tal proposito il W3C ha definito le due modalità illustrate nei paragrafi seguenti.

DTD (*Document Type Definition*)

Si tratta di un particolare documento che definisce i tag utilizzabili in un documento XML e anche la loro struttura. Il seguente esempio mostra un DTD per il linguaggio XML di definizione di libri usato nel documento visto in precedenza

```
<!ELEMENT todolist (item)+>
<!ELEMENT item (number,priority,description,state)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT state (#PCDATA)>
```

Il DTD dichiara che il tag `todolist` contiene un numero non precisato di elementi `item`; questo a sua volta contiene i tag `number`, `priority`, `description` e `state` che contengono solo caratteri.

Schema

Rappresenta un modo alternativo ai DTD per esprimere la struttura di un documento. Il principale vantaggio degli Schema consiste nel fatto che vengono essi stessi descritti in XML. Il seguente esempio è lo schema per la definizione del documento XML precedente.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
```

```
   <xsd:element name="todolist" type="TodoListType"/>

   <xsd:complexType name="ItemType">
      <xsd:element name="item" type="ItemType" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:complexType>

   <xsd:element name="item" type="ItemType"/>

   <xsd:complexType name="ItemType">
      <xsd:element name="number" type="xsd:integer"/>
      <xsd:element name="priority" type="xsd:integer"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="state" type="xsd:integer"/>
   </xsd:complexType>

</xsd:schema>
```

Il documento precedente sarà contenuto in un file `.xsd` (XML Schema Definition). L'elemento principale è `schema` che a sua volta contiene altri elementi: `element` definisce un elemento del documento XML e le informazioni ad esso associato (numero di occorrenze, …); `complexType` definisce un tipo composto, ovvero un elemento che contiene a sua volta altri elementi e può avere attributi. Maggiori informazioni sugli Schema si possono trovare in [2].

Un documento XML in sé è inutile se non viene elaborato da un programma e, in primo luogo, da un parser. I parser XML possono essere *non validanti* oppure *validanti*. Mentre i primi verificano che il documento ricevuto in input sia ben formato, i parser validanti verificano anche la struttura del documento basandosi sul corrispondente DTD o schema. Di seguito si approfondiranno gli aspetti di elaborazione dei documenti XML.

Elaborazione di documenti XML

Nei paragrafi precedenti si sono approfondite le varie tecnologie alla base di XML. A questo punto si devono mettere in pratica i concetti visti cominciando a parlare di elaborazione di documenti XML.

Dato un documento XML questo sarà ragionevolmente elaborato da un'applicazione per vari scopi (eseguire calcoli sui dati contenuti, visualizzarli in una pagina Web, trasmetterli a un altro programma, …). Alla base di tutto ci sarà quindi un parser XML, come visto in precedenza, per assicurarsi che il documento sia ben formato e valido. Sorge ora il problema di come usare i dati contenuti all'interno del documento; non è infatti pensabile di ragionare in termini di elaborazione di stringhe ed è quindi necessaria una API che permetta di elaborare il documento a un livello di astrazione più elevato. Lo standard prevede quanto segue.

SAX (*Simple API for XML*)

Si tratta di una semplice API *event driven* sviluppata originariamente in Java dal suo autore [3].

DOM (*Document Object Model*)

Framework che permette di trasformare un documento XML in una struttura gerarchica più facilmente maneggevole [4].

Nel mondo Java si sta facendo strada un'alternativa a DOM, JDOM, una API che offre un modello a oggetti più semplice rispetto a quello offerto da DOM. Si osservi comunque che JDOM non è uno standard W3C ma una API proprietaria.

È necessario precisare che i tool per elaborare i documenti XML, e quindi anche i parser, sono in linea di principio indipendenti dal linguaggio e dalla piattaforma. Ad esempio Microsoft fornisce un proprio parser che supporta SAX e DOM. Alcuni fra i principali parser realizzati in Java sono i seguenti.

XML4J

La tecnologia alla base del parser XML di IBM è stata donata all'Apache Group per lo sviluppo del parser Xerces (`http://www.alphaworks.ibm.com/tech/xml4j`).

Xerces

È un parser sviluppato dal team di Apache all'interno dell'Apache XML Project. È basato sui sorgenti di XML4J, implementa DOM (livello 1 e 2) e SAX (versione 2) e ha un supporto preliminare per gli schema XML (`http://xml.apache.org/xerces-j`). È disponibile inoltre una versione in C++ con i wrapper Perl e COM.

JAXP (*Java API for XML Parsing*)

Si tratta del parser fornito da Sun, precedentemente conosciuto come Project X. Supporta SAX e DOM e inoltre espone una API che permette di integrare parser di terze parti.

XP

XP è un parser non validante conforme alle specifiche 1.0 di XML. Oltre a una normale API ad alto livello fornisce anche una API a basso livello per sviluppare nuovi parser (`http://www.jclark.com/xml/xp`).

Gli esempi presentati nel corso del capitolo faranno riferimento al parser Xerces.

SAX

SAX (Simple API for XML) è un'interfaccia event driven per l'elaborazione dei documenti XML giunta al momento alla versione 2. Il supporto per la API SAX si trova nel package `org.xml.sax`; i package `org.xml.sax.ext` e `org.xml.sax.helpers` forniscono alcune classi accessorie.

A differenza di DOM, che crea la struttura ad albero del documento residente in memoria, i parser SAX generano eventi che vengono intercettati da un apposito handler fornito dallo sviluppatore. Più precisamente il parser SAX legge il documento XML e genera eventi corrispondenti alla struttura del documento. Lo sviluppatore deve definire un handler, che implementa l'interfaccia `org.xml.sax.ContentHandler`, per definire i metodi callback che vengono invocati in corrispondenza degli eventi. In SAX 1 si usava invece l'interfaccia `DocumentHandler`; `ContentHandler` aggiunge a questa il supporto ai namespace e la notifica di entità tralasciate durante il parsing non validante.

Si vedranno ora in dettaglio i metodi principali dell'interfaccia `ContentHandler`. L'inizio e la fine del parsing di un documento vengono intercettati dai metodi `startDocument()` ed `endDocument()` utilizzati ripettivamente per inizializzare l'handler, se necessario, e per eseguire particolari azioni al termine dell'output, come clean-up o generazione di un particolare output per comunicare l'esito del parsing. Il metodo `endDocument()` non viene comunque richiamato se il parser ha incontrato un errore: in tal caso infatti il parsing viene interrotto e si generano opportuni eventi di gestione degli errori. I metodi più interessanti di `ContentHandler` sono quelli di gestione dei tag `startElement()` ed `endElement()`, chiamati rispettivamente quando il parser incontra un tag aperto (`<FOO>`) e un tag chiuso (`</FOO>`). Per poter elaborare le informazioni contenute nel tag il metodo `startElement()` ha come parametri: l'URI associato al namespace, il nome locale dell'elemento, il nome qualificato dell'elemento, gli attributi associati al tag rappresentati come un oggetto `Attributes`.

La firma del metodo `startElement()` è la seguente

```
public void startElement(String URI, String lName, String qName,
                         Attributes attr) throws SAXException
```

I parametri permettono di identificare i vari componenti di un elemento, comprese le informazioni sui namespace. Si supponga ad esempio di avere il seguente tag

```
<list:item xmlns:list="http://www.foo.com/ns">
```

In questo caso il namespace URI è `http://www.foo.com/ns` mentre il nome locale è `item`. Se il parser è configurato per l'elaborazione dei prefissi allora i metodi `startElement()` ed `endElement()` riportano `list:item` come nome qualificato altrimenti tale parametro potrebbe non essere valorizzato.

Si noti che, come accennato in precedenza, il supporto ai namespace è stato introdotto con la versione 2.0 delle specifiche; il corrispondente metodo dell'interfaccia `DocumentHandler` aveva come parametri solo il nome dell'elemento e la lista di parametri.

Si osservi che i tag di un documento XML possono essere arbitrariamente annidati ma il metodo `startElement()` non comunica alcuna informazione relativa al contesto in cui un elemento viene incontrato. È quindi onere del programmatore tenere traccia delle sequenze dei tag (usando ad esempio uno stack).

Il metodo `characters()` viene invocato per comunicare all'handler il contenuto del documento compreso fra due tag. I dati vengono passati come un array di byte: sarà il programmatore a convertirlo eventualmente in una stringa. Per semplificare il compito dei programmatori viene fornita la classe `DefaultHandler`, definita nel package `org.xml.sax.helpers`, che implementa tutti i metodi di `ContentHandler` come no-operation; i metodi sono cioè vuoti. In questo modo per definire un proprio handler è sufficiente estendere `DefaultHandler` e ridefinire solo i metodi necessari.

Si consideri ora il documento XML visto in precedenza

```
<?xml version="1.0"?>
<todolist>
   <item>
      <number>1</number>
      <priority>6</priority>
      <description>Leggere la posta</description>
      <state>2</state>
   </item>
   ...
</todolist>
```

La tab. 13.1 mostra la corrispondenza fra gli elementi del documento e gli eventi SAX.

Tabella 13.1 – *Corrispondenza fra elementi del documento ed eventi SAX generati*

Elementi	Eventi SAX
`<todolist>`	`startDocument()`
`<item>`	`startElement()`
`<number>`	`startElement()`
`1`	`characters()`
`</number>`	`endElement()`
`<priority>`	`startElement()`
`6`	`characters()`
`</priority>`	`endElement()`
...	...
`</item>`	`endElement()`
...	...
`</todolist >`	`endDocument()`

Il codice seguente mostra come istanziare il parser SAX di Xerces ed eseguire il parsing di un documento.

```
import org.apache.xerces.parsers.*;
...

SAXParser parser = new SAXParser();

parser.setContentHandler(new MyContentHandler());
parser.setErrorHandler(new MyErrorHandler());

String file_name = "file:" + new File("test.xml").getAbsolutePath();
parser.parse(new InputSource(file_name));
```

Per eliminare ogni riferimento alla specifica implementazione del parser è possibile utilizzare JAXP e ottenere un parser nel modo seguente

```
import javax.xml.parsers.*;

SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setValidating(true);
SAXParser sp = spf.newSAXParser();
Parser parser = sp.getParser();

...
```

Poichè JAXP non supporta al momento SAX 2 sarebbe necessario utilizzare l'interfaccia `DocumentHandler`.

L'input del parser è un'istanza della classe `org.xml.sax.InputSource` che incapsula appunto la sorgente dati di un documento XML in questo caso specificata attraverso un system ID, cioè un URI a un file.

Dato il documento XML precedente contenente la todo-list si supponga di voler contare il numero di attività. Si supponga ad esempio di voler contare il numero di elementi nella todo-list; la seguente classe `ItemCounter` esegue questo compito

```
public class ItemCounter extends DefaultHandler {
   private int counter = 0;

   public void startElement(String uri, String name, String qName,
                            Attributes attributes) throws SAXException {
      if (qName.equals("item")) {
         counter++;
      }
   }

   public void endDocument() {
```

```
      System.out.println("Trovate " + counter + " attività");
   }
}
```

Analogamente, anche per la gestione degli errori si prevede un apposito handler che implementa l'interfaccia `org.xml.sax.ErrorHandler`; tale interfaccia definisce i seguenti metodi

`warning()`

Segnalazione di un warning.

`error()`

Errore recuperabile.

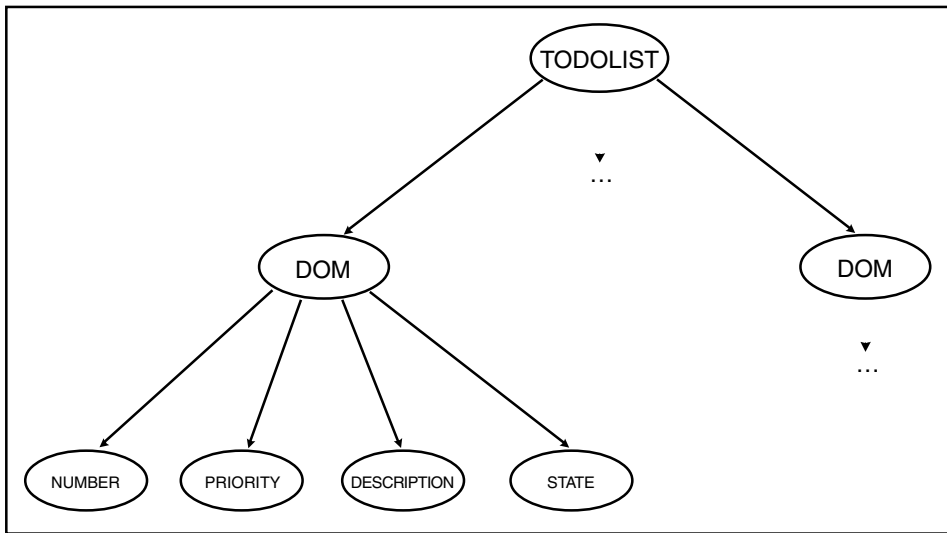
`fatalError()`

Errore non recuperabile.

La classe `DefaultHandler` implementa comunque anche i metodi di `ErrorHandler` e può essere estesa per definire un gestore di errori custom. Il parser può quindi essere configurato per segnalare gli errori non tramite eccezioni al programma client ma attraverso l'interfaccia `ErrorHandler`. La seguente implementazione di esempio ridefinisce i metodi `error()` e `warning()`.

```
public class MyErrorHandler extends DefaultHandler {
   public void error(SAXParseException e) throws SAXException {
      throw e;
   }

   public void warning(SAXParseException e) throws SAXException {
      System.out.println(e.getMessage());
   }
}
```

SAX è una API molto pratica per semplici elaborazioni di documenti XML. A volte un documento ha però una struttura troppo complessa per essere gestita con SAX perché il codice dell'handler diventerebbe troppo complicato. Per risolvere questi problemi si può usare DOM, che sarà approfondito nei prossimi paragrafi.

Figura 13.1 – *Rappresentazione DOM di una struttura XML*

DOM

DOM è un modello a oggetti standard per la gestione di documenti XML, HTML e, in prospettiva, WML. La principale differenza rispetto a un parser SAX consiste nel fatto che l'elaborazione in SAX è sequenziale e avviene nel corso del parsing. Le informazioni sul documento non sono più disponibili al termine dell'elaborazione a meno che il programmatore non si sia costruito una struttura ausiliaria. DOM svolge proprio questa funzione, ovvero esegue il parsing del documento XML e restituisce una struttura gerarchica ad albero che lo rappresenta; dopo l'esecuzione del parsing è quindi possibile elaborare più volte la stessa struttura.

Il W3C fornisce la definizione delle interfacce DOM mediante l'IDL di CORBA ma rende disponibile anche il binding con Java che è contenuto nel package `org.w3c.dom`.

Riprendendo la todo-list precedente si ha la rappresentazione DOM mostrata in fig 13.1.

DOM identifica: le interfacce e gli oggetti usati per la rappresentazione dei documenti, la semantica di tali interfacce e oggetti, relazioni e collaborazioni fra oggetti e interfacce.

DOM è stato sviluppato all'interno del W3C e le specifiche sono suddivise nelle due parti illustrate nei paragrafi seguenti.

DOM Level 1

Definisce le interfacce principali per la manipolazione dei documenti. Questa parte delle specifiche è allo stato di Recommendation W3C e si compone di

- Core: definisce le funzionalità di base per la gestione dei documenti
- HTML: per la gestione di documenti HTML

DOM Level 2

Introduce moduli opzionali per funzionalità avanzate come viste, stylesheet, eventi, attraversamento della struttura di un documento e range di contenuti all'interno di un documento. Attualmente è allo stato di Candidate Recommendation W3C.

Al momento sono stati definiti i requisiti di DOM Level 3 che definirà, fra le altre cose, spostamento di un nodo fra documenti e ordinamento di nodi, estensione del modello di eventi di DOM Level 2, content model e validation use case, caricamento e salvataggio di un documento XML, gestione delle viste e formattazione.

Si approfondiranno ora le interfacce principali di DOM che, nel binding Java, sono contenute nel package `org.w3c.dom`. Dove necessario si specificheranno le differenze fra Level 1 e Level 2. Come accennato in precedenza un documento viene rappresentato come un insieme di nodi ed è quindi naturale che l'interfaccia principale sia `Node`. Questa rappresenta un generico nodo nella struttura gerarchica di un documento e definisce i metodi per accedere ad altri nodi nel documento. Particolari tipi di nodi sono rappresentati da interfacce che estendono `Node`; segue ora una lista delle interfacce principali.

`Attr`

Rappresentazione di un attributo in un elemento. Questo è un caso particolare di nodo in quanto pur estendendo l'interfaccia `Node` un attributo non viene considerato dal DOM come nodo figlio dell'elemento che descrive. Si ha quindi che i metodi che accedono ai nodi dell'albero (come `getChildNodes()`) ritornano `null`.

`CharacterData`

Rappresenta una sequenza di caratteri all'interno del documento ed espone metodi per la gestione di stringhe. Sono definite tre interfacce che la estendono: `Text`, `CDATASection` e `Comment`.

`Text`

Questa interfaccia estende `CharacterData` e rappresenta il testo contenuto all'interno di un elemento o di un attributo. Se il testo non contiene marcatori allora è compreso

all'interno di un unico nodo `Text` altrimenti ne viene eseguito il parsing e lo si inserisce in una lista di nodi.

`CDATASection`

Interfaccia che estende `Text` e rappresenta un nodo che può contenere testo organizzato con marcatori.

`Comment`

Estende `CharacterData` e rappresenta il contenuto di un commento contenuto fra i caratteri `<!--` e `-->`.

`Element`

Interfaccia per la gestione di un elemento e dei corrispondenti attributi.
Si consideri ad esempio il seguente documento XML

```
<ROOTELEMENT>
   <ELEMENT1>
   </ELEMENT1>
   <ELEMENT2>
         <SUBELEMENT1>
      </SUBELEMENT1>
   </ELEMENT2>
</ROOTELEMENT>
```

Il corrispondente documento DOM avrà un elemento radice `ROOTELEMENT`, quindi due elementi child `ELEMENT1` ed `ELEMENT2`; quest'ultimo a sua volta ha un elemento child `SUBELEMENT1`.

Un'altra fondamentale interfaccia del DOM è `Document` che, come accennato in precedenza, rappresenta l'intero documento XML. L'interfaccia `Document` fornisce inoltre i metodi factory per creare gli altri oggetti che compongono il documento come elementi e nodi di testo. Per la documentazione delle altre interfacce si rimanda a [4].
Si vedrà ora un esempio di creazione di un documento XML utilizzando DOM.

```
Document doc = new DocumentImpl(null);
// Il codice che segue è indipendente dal particolare parser utilizzato
```

```
// Creazione dell'elemento root del documento, identificato dal tag <DOCUMENT>
Element root = doc.createElement("DOCUMENT");
doc.appendChild(root);

// Creazione e inserimento di un nodo
Element element1 = doc.createElement("CHAPTER1");
root.appendChild(element1);

// Creazione e inserimento di un nodo discendente di <ELEMENT1>
element1.appendChild(doc.createElement("PARAGRAPH1"));

// Creazione e inserimento di un secondo nodo
// discendente di <DOCUMENT>
root.appendChild(doc.createElement("CHAPTER2"));
```

La prima linea di codice è l'unica dipendente dalla particolare implementazione del parser mentre il codice seguente usa la API standard DOM. Per creare un elemento si usa l'apposito metodo factory `createElement()` definito nell'interfaccia `Document`, quindi lo si collega al resto dell'albero con `appendChild()`.

Il seguente codice mostra come attraversare i nodi dell'albero mediante un algoritmo depth-first.

```
void printDocument1(Node node, int level, Writer writer) {

   try {

      // indentazione
      for (int i = 0; i <= level; i++) {
         writer.write(" ");
      }

      // stampa del contenuto del nodo
      writer.write(node.getNodeName());
      writer.write("\n");
      writer.flush();

      // per ogni nodo figlio si richiama ricorsivamente
      // questa funzione
      NodeList list = node.getChildNodes();
      for(int i = 0; i < list.getLength(); i++) {
         printDocument1(list.item(i), level+1, writer);
      }
   } catch (IOException e) {
      System.err.println("IO exception: " + e.getMessage());
   }
}
```

Con DOM Level 2 sono state introdotte delle nuove interfacce per il supporto alla navigazione dei documenti DOM le quali, nelle implementazioni Java, sono incluse nel package `org.w3c.dom.traversal`. È possibile ad esempio utilizzare l'interfaccia `NodeIterator` che permette di scorrere un insieme di nodi attraverso il metodo `nextNode()`. Il codice precedente potrebbe quindi essere riscritto nel modo seguente

```
void printDocument2(NodeIterator iterator) {

   Node node;

   System.out.println();
   while (true) {
      try {
         node = iterator.nextNode();
      } catch (DOMException e) {
         System.err.println("DOM Exception: " + e.getMessage());
         return;
      }
      if (node == null) {
         iterator.detach();
         return;
      }
      System.out.println(node.getNodeName());
   }

}
```

La creazione di un iterator avviene mediante una classe factory che deve implementare l'interfaccia `org.w3c.dom.traversal.DocumentTraversal` e il corrispondente metodo `createNodeIterator()`. Prendendo come riferimento Xerces, viene fornita una classe — `org.apache.xerces.dom.DocumentImpl` — che implementa l'interfaccia `Document` e anche `DocumentTraversal`. Il metodo precedente può quindi essere richiamato con

```
NodeIterator iterator
= ((DocumentImpl)doc).createNodeIterator(root, NodeFilter.SHOW_ALL, null, false)

printDocument2( iterator );
```

Il metodo `createNodeIterator()` accetta i seguenti parametri:

- il nodo da cui partire con l'iterazione
- un flag che indica quali tipi di nodo far restituire all'iteratore. L'interfaccia `org.w3c.dom.traversal.NodeFilter` definisce una serie di opportune co-

stanti eventualmente componibili in OR. Nell'esempio precedente di richiede di mostrare tutti i nodi

- il particolare filtro da applicare all'iterazione, rappresentato dall'interfaccia `NodeFilter`, oppure `null` se non si vuole applicare alcun filtro
- un flag booleano che indica se espandere o meno i riferimenti a entità

DOM è una API molto potente per elaborare documenti XML e rappresenta la soluzione ideale nella maggior parte dei casi; ciò nonostante DOM non è esente da critiche.

Come visto in precedenza i parser DOM devono mantenere in memoria l'intero documento XML e per documenti con una grande mole di dati questo potrebbe rappresentare un problema. In realtà è sufficiente che i client "abbiano l'impressione" che l'intero documento sia residente in memoria ma per i parser non si ha alcun vincolo implementativo; ad esempio è possibile implementare una strategia lazy che carica i nodi su richiesta, come viene fatto da Xerces.

Gli oggetti DOM non permettono poi di incapsulare logica applicativa; non è possibile quindi scrivere in DOM codice fortemente tipizzato di questo tipo

```
String code = customer.getCode();
```

Questo porta ad avere oggetti DOM e oggetti applicativi (ad esempio EJB) distinti, con la necessità di disporre di metodi di utilità per caricare una collezione di EJB a partire da una sorgente XML. Una successiva sezione approfondirà questa soluzione.

Un'altra limitazione di DOM è data dalla complessità della API che a volte è eccessivamente verbosa. Una possibile soluzione consiste nello sviluppo di uno strato software al di sopra di DOM che fornisca una API più pratica e con un miglior livello di astrazione. La successiva sezione descrive JDOM, un progetto open source che rappresenta un tentativo in tal senso.

JDOM

Nei paragrafi precedenti sono stati trattati SAX e DOM, i due standard per la gestione di documenti XML. Essendo standard si tratta quindi di librerie indipendenti dal linguaggio utilizzato. In questo paragrafo si introdurrà JDOM [5], una API sviluppata appositamente per gestire documenti XML in Java.

JDOM è una API open source che sfrutta le caratteristiche di Java come overloading e Collection API. L'obiettivo principale di JDOM è mettere a disposizione dello sviluppatore Java una libreria potente ma allo stesso tempo con requisiti minimi per quanto riguarda l'occupazione di memoria.

Si osservi che nonostante JDOM sia rivolta agli sviluppatori Java, è comunque una API che interagisce con SAX e DOM e quindi i programmi sviluppati con JDOM possono ricevere documenti XML da un parser SAX/DOM e analogamente possono generare un documento DOM o uno stream di eventi SAX.

Un documento XML è rappresentato come un'istanza della classe `org.jdom.Document` e può essere creato da varie sorgenti dati attraverso i builder. Sono definiti due tipi di builder, `DOMBuilder` e `SAXBuilder`. Come è evidente dal nome, `DOMBuilder` carica un documento a partire da un oggetto `org.w3c.dom.Document` mentre `SAXBuilder` sfrutta un parser SAX. È comunque possibile sviluppare dei builder che generino un `Document` a partire ad esempio da query SQL o altre sorgenti dati. Il concetto di builder permette cioè di incapsulare la generazione del documento XML/JDOM e la sua sorgente. Il seguente codice mostra come istanziare un documento JDOM sfruttando un parser SAX.

```
SAXBuilder builder = new SAXBuilder(true);
Document doc = builder.build(new File(args[0]));

// ...
```

La classe `SAXBuilder` ha vari costruttori in overloading che permettono di specificare quale parser usare (il default è Xerces) e se eseguire o meno la validazione del documento (il default è `false`).

JDOM mette a disposizione alcune classi per passare all'esterno del programma un documento XML. Ad esempio la classe `XMLOutputter` scrive un documento JDOM su un particolare stream come si può vedere dal seguente codice.

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(doc, System.out);
```

In alternativa si possono usare `SAXOutputter` e `DOMOutputter`. Il primo genera una sequenza di eventi SAX che possono essere intercettati da un'applicazione, mentre il secondo genera un documento DOM.

Si supponga ora di voler costruire un documento JDOM in modo programmatico. A differenza di SAX e DOM non è necessario usare metodi factory; ad esempio il frammento di codice

```
Element root = new Element("ROOT");
Document doc = new Document(root);

root.setContent("Documento creato con JDOM");
```

produrrà il seguente documento

```
<ROOT>Document created with JDOM</ROOT>
```

Per aggiungere elementi si usa il metodo `addChild()`.

```
Element child1 = new Element("CHILD1").setContent("First child");
Element child2 = new Element("CHILD2").setContent("Second child");

root.addChild(child1);
root.addChild(child2);
```

Nel caso in cui si vogliano creare dei template per particolari elementi del documento è sufficente estendere la classe `Element`. Se ad esempio si vuole creare un elenco di pianeti, potrebbe essere utile un elemento che aggiunga automaticamente i tag `<PLANET> </PLANET>` in modo che il codice

```
root.addChild(new PlanetElement("Mars"));
```

generi il seguente nodo

```
<PLANET>Mars</PLANET>
```

Il codice della corrispondente classe `PlanetElement` risulta molto semplice.

```
public class PlanetElement {
   public PlanetElement(String name) {
      super("PLANET");
      setContent(name);
   }
}
```

Dato un documento XML è possibile accedere all'elemento radice con il seguente codice

```
Element root = doc.getRootElement();
```

A questo punto si può accedere alla lista di tutti i nodi figli

```
List childrenList = root.getChildren();
```

o a tutti i nodi figli aventi lo stesso nome

```
List childrenList = root.getChildren("CHAPTER");
```

Si osservi che i metodi precedenti restituiscono un oggetto `List` della Collection API di Java. Le istanze di `List` (o eventualmente di `Map`) restituite dai metodi di JDOM sono aggiornabili e le modifiche si ripercuotono sul `Document` corrispondente. Ad esempio con

```
childrenList.add(new Element("PARAGRAPH"));
```

si aggiunge un nodo discendente all'elemento `CHAPTER`. Per accedere al primo nodo figlio con un particolare nome si usa il seguente metodo

```
Element e = root.getChild("CHAPTER");
```

Per accedere al contenuto di un elemento

```
String content = e.getContent();
```

Utilizzando DOM si sarebbe invece dovuto scrivere

```
String content = e.getFirstChild().getNodeValue();
```

Nel caso in cui l'elemento abbia un attributo lo si può ottenere come stringa o come istanza della classe `Attribute`

```
String attrValue = e.getAttributeValue("name");

Attribute attribute = e.getAttribute("name");
String value = attribute.getValue();
```

È inoltre possibile accedere al valore tipizzato di un attributo mediante una serie di metodi specifici. Ad esempio, per ottenere il valore di un attributo numerico si può scrivere

```
int length = attribute.getAttribute("length").getIntValue();
```

Per impostare il valore di un attributo si scriverà invece

```
e.setAttribute("height", "20");
```

JDOM presenta altre caratteristiche avanzate come supporto ai namespace e gestione di vari tipi di elementi (processing instruction, commenti, ...).

JDOM è sicuramente uno strumento interessante per lo sviluppatore Java anche se al momento è una API ancora immatura. Il vantaggio principale che presenta non è tanto la disponibilità di servizi avanzati, che si trovano anche nei comuni parser, quanto il livello di astrazione che fornisce allo sviluppatore.

Java XML binding

Nei paragrafi precedenti si sono esaminate alcune API per elaborare documenti XML, prevalentemente basandosi su stream di eventi e strutture ad albero. Una possibile alternativa consiste nel *data binding*, ovvero nel convertire un documento XML in un oggetto

Java e viceversa. Al momento esiste una Java Specification Request (JSR-031) [6] sviluppata da Sun per la realizzazione di supporto per data binding XML. Tale supporto sarà costituito da uno schema compiler, che traduce uno schema XML in un insieme di classi, e da un framework di marshalling che permette di eseguire l'unmarshalling di un documento XML in un grafo di oggetti e il corrispondente marshalling del grafo in un documento XML.

Riprendendo l'esempio della precedente todo-list si ha che un ipotetico schema compiler genererebbe codice simile al seguente

```
public class Item {
   private int number;
   private int priority;
   private String description;
   private int state;

   public Item() {}

   public int getNumber() {return number;}
   public void setNumber(int number) {this.number = number;}
   public int getPriority() {return priority;}
   public void setPriority(int priority) {this.priority = priority;}
   public String getDescription() {return description;}
   public void setDescription(String description) {
      this.description = description;
   }
   public int getState() {return state;}
   public void setState(int state) {this.state = state;}

   public void marshal(OutputStream out) throws IOException;
   public static Item unmarshal(InputStream in) throws IOException;
}
```

In attesa di un'estensione della piattaforma Java per il data binding vi sono alcuni tool e framework che supportano queste funzionalità come ad esempio Enhydra [7], KaanBaan [8] e XMLBreeze [9].

Che cosa scegliere?

Come si è visto nel corso di queste sezioni, per realizzare applicazioni XML il programmatore ha a sua disposizione più alternative.

Si può usare una API lightweight come SAX, adatta per elaborazioni in cui non sia necessario mantenere in memoria l'intero documento XML. D'altro canto, per applicazioni più complesse, DOM risulta sicuramente più adeguato, mettendo a disposizione del programmatore una struttura gerarchica senza la necessità di dover scrivere codice addizionale.

Considerando Java come specifico linguaggio di programmazione, entra allora in gioco anche JDOM che, come visto, mette a disposizione una API molto semplice e meno verbosa di DOM, oppure l'utilizzo del data binding che offre il grosso vantaggio di utilizzare oggetti Java fortemente tipizzati.

XSL

Nella sezione introduttiva su XML si è accennato a XSL, il linguaggio usato per trasformare i documenti XML. In questa sezione si approfondiranno i concetti visti.

XSL [10] è composto dalle due parti illustrate di seguito.

- XSLT (*XSL Transformation*): Un linguaggio per trasformare documenti XML in altri documenti. Si noti che il formato di destinazione potrebbe essere ancora XML oppure qualcosa di diverso, come ad esempio un formato grafico. In questo caso il foglio di trasformazione viene chiamato stylesheet. Questo è comunque solo un esempio dei possibili utilizzi di XSLT.

- FO (Formatting Objects): Un linguaggio per descrivere il layout del testo. Un documento XSL:FO contiene quindi delle direttive per indicare al motore di rendering come visualizzare il documento. Per maggiori informazioni si rimanda a [11].

Si noti che "trasformazione di documenti" è un'espressione generica che comprende il mapping di un documento XML in una pagina HTML ma anche la trasformazione di un documento in un dato schema in uno schema diverso. XSL è quindi alla base delle principali applicazioni XML, ovvero Web Publishing e integrazione di applicazioni. Un documento XSL viene elaborato da un processore che, dati in ingresso un documento XML e un documento XSL, restituisce un nuovo documento XML. Si vedranno ora alcuni fondamenti di XSLT. Anzitutto, una trasformazione XSL viene definita in un file XML come segue

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
...
</xsl:stylesheet>
```

L'intestazione del documento indica la versione di XSL utilizzata e il namespace utilizzato, ovvero si dichiara che si useranno dei tag con prefisso `xsl` e che la definizione di questi tag corrisponde all'URL indicato. È necessario porre molta attenzione al namespace poiché engine XSL diversi potrebbero implementare namespace diversi e incompatibili.

XSLT è un linguaggio dichiarativo; una trasformazione viene quindi specificata con una regola che attraverso un meccanismo di pattern matching viene applicata all'elemento XML di partenza. Una regola viene espressa mediante un template.

```
<xsl:template match="node">
...
</xsl:template>
```

Quando l'engine XSL incontrerà questa regola allora ricercherà nel documento XML un elemento di nome `node` e ad esso applicherà la trasformazione. Per attivare la regola precedente è necessario utilizzare il comando

```
<xsl:apply-templates select="node"/>
```

Si consideri ora il seguente documento XML

```
<?xml version="1.0"?>
<article>
<title>XSL and Java</title>
<author email="doe@doe.com">John Doe</author>
</article>
```

Per accedere al contenuto di un singolo nodo si usa il comando `xsl:value-of`; il contenuto del nodo `title` sarà quindi

```
<xsl:value-of select="title"/>
```

Il valore dell'attributo `select` permette molta libertà nell'accesso agli elementi. È ad esempio possibile accedere agli attributi di un elemento usando il carattere `@`; il valore dell'attributo `email` di `author` è quindi

```
<xsl:value-of select="author/@email"/>
```

Si possono inoltre specificare elementi annidati (`"article/author"`), wildcards (`*/title`) e altri pattern mediante espressioni XPath [12]. XPath è un linguaggio definito dal W3C per individuare le singole parti di un documento XML ed è al momento giunto alla versione 1.0.

Vi sono molti altri comandi in XSLT. Si possono ad esempio esprimere condizioni con i comandi `xsl:choose` e `xsl:if`. Il primo permette una scelta multipla fra diverse condizioni, analogamente allo `switch` in Java

```
<xsl:choose>
   <xsl:when test="condition1">
      ...
   </xsl:when>
   <xsl:when test="condition2">
```

```
      ...
   </xsl:when>
   <xsl:otherwise>
      ...
   </xsl:otherwise>
</xsl:choose>
```

mentre `xsl:if` permette di indicare una sola condizione e nessuna alternativa

```
<xsl:if test="condition">
   ...
</xsl:if>
```

In XSLT è inoltre possibile esprimere cicli con l'istruzione `xsl:for-each`

```
<xsl:for-each select="pattern">
</xsl:for-each>
```

Si vedrà ora un esempio completo di utilizzo di XSL. Sia dato il seguente documento, una lista di articoli.

```
<?xml version="1.0" ?>
<articles>
   <article>
      <title>XSL and Java</title>
      <author id="1">
         <first-name>John</first-name>
         <last-name>Doe</last-name>
         <email>doe@doe.com</email>
      </author>
   </article>
   <article>
      <title>Distributed Java programming</title>
      <author id="2">
         <first-name>Tod</first-name>
         <last-name>Mann</last-name>
         <email>tod@foo.com</email>
      </author>
   </article>
   <article>
      <title>Java, XML and enterprise application integration</title>
      <author id="3">
         <first-name>Frank</first-name>
         <last-name>Kipple</last-name>
         <email>kipple@bar.com</email>
      </author>
   </article>
</articles>
```

Il documento precedente potrebbe essere il risultato di una query su database; per trasformarlo in una tabella HTML in modo da visualizzarla in un browser si userà il seguente documento XSL.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl
= "http://www.w3.org/1999/XSL/Transform" version="1.0">

   <xsl:template match="/">
      <HTML>
      <BODY>
         <TITLE>Articles' list</TITLE>
         <TABLE BORDER="1">
            <TH>Title</TH><TH>Author</TH><TH>email</TH>
            <xsl:apply-templates select="//article"/>
         </TABLE>
      </BODY>
      </HTML>
   </xsl:template>

   <xsl:template match="article">
      <TR>
         <TD><x<xsl:template match="article"></TD>
         <TD><xsl:value-of select="title"/></TD>
         <TD><xsl:value-of select="author/first-name"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="author/last-name"/>
         </TD>
         <TD><xsl:value-of select="author/email"/></TD>
      </TR>
   </xsl:template>

</xsl:stylesheet>
```

Per visualizzare la pagina HTML generata dal precedente file XSL ci sono due possibilità:

- la prima è trasferire i file XML e XSL direttamente al browser, ma si tratta di una funzionalità supportata al momento solo da Internet Explorer.

- la seconda consiste nell'eseguire la trasformazione sul server e mandare al browser una comune pagina HTML. In questo modo si garantisce il supporto a più browser e, parametrizzando opportunamente la trasformazione XSL, è possibile ad esempio generare codice WML per i browser WAP.

Si vuole ora realizzare come esempio un semplice Servlet che genera il codice HTML corrispondente ai file XML e XSL specificati come parametri. Il metodo `doGet()` del

Servlet, chiamato `XMLServlet`, inoltrerà la richiesta al seguente metodo `doXSLProcessing()`

```
private void doXSLProcessing(HttpServletRequest request, HttpServletResponse response)
                              throws IOException, ServletException {

   response.setContentType("text/html");

   PrintWriter out = response.getWriter();

   try {

      // Si ottengono gli URL corrispondenti ai
      // documenti XML e XSL specificati come parametri.
      String xmlUrl = getDocURL("XML", request);
      String xslUrl = getDocURL("XSL", request);

      XSLTProcessor proc = XSLTProcessorFactory.getProcessor();

      // Si esegue la trasformazione del documento
      // dirigendone l'output sul browser
      proc.process(new XSLTInputSource(xmlUrl),
                   new XSLTInputSource(xslUrl),
                   new XSLTResultTarget(out));

   } catch(org.xml.sax.SAXException e) {
      out.write(e.getMessage());
   } finally {
      proc.reset();
   }
}
```

Il metodo, ottenuti gli URL del documento XML da visualizzare e del file XSL da usare per la formattazione, manda il risultato della trasformazione al browser usando Xalan. Xalan incapsula il comportamento dell'engine XSL nell'interfaccia `XSLTProcessor`; si osservi che in questo modo non si fa riferimento a un'implementazione specifica. I dettagli relativi alla creazione dell'engine vengono incapsulati nella classe factory `XSLTProcessorFactory`. Il metodo `process()` di `XSLTProcessor` esegue la trasformazione vera e propria del documento e ha la seguente firma

```
process(XSLTInputSource xmlSource, XSLTInputSource xslStylesheet,
      XSLTResultTarget resultTree) throws org.xml.sax.SAXException;
```

Il parametro `xmlSource` rappresenta il documento XML da trasformare mentre `xslStylesheet` è lo stylesheet XSL da usare per la trasformazione. Il documento risul-

Figura 13.2 – *Risultato della trasformazione XSL applicata dal Servlet*

Title	Author	email
XSL and Java	John Doe	doe@doe.com
Distributed Java programming	Tod Mann	tod@foo.com
Java, XML and enterprise application integration	Frank Kipple	kipple@bar.com

tato della trasformazione è rappresentato dal parametro `resultTree`. Le classi `XSLTInputSource` e `XSLTResultTarget` sono dei contenitori per i documenti coinvolti nella trasformazione e possono essere istanziate a partire da un URL, uno stream, un nodo DOM o un handler SAX.

Per attivare il Servlet si digita il seguente URL

```
http://localhost/xsl/servlet/XMLServlet?XML=articles.xml&XSL=list.xsl
```

e verrà visualizzata la pagina illustrata in fig. 13.2.

La trasformazione di documenti XML è alla base delle applicazioni che vogliano sfruttare questa tecnologia. Il vantaggio principale di usare XSL consiste nel non dover cablare la logica di trasformazione nel codice. In questo modo è possibile apportare modifiche senza dover ricorrere a ricompilazioni dell'applicazione. Per approfondire l'utilizzo di XSL e XPath con Java si veda [13].

Publishing di documenti

Negli ultimi tempi si è avuto un crescente utilizzo di applicazioni web o comunque di client web per utilizzare applicazioni legacy esistenti. Si è quindi assistito all'introduzione di nuove figure come i web designer e alla necessità di separare l'elaborazione dei dati dai dettagli della loro visualizzazione; in questo modo si ha una completa separazione delle responsabilità e i programmatori non devono diventare esperti di grafica web così come i web designer non devono imparare a programmare.

Java si inserisce in questo quadro con i Servlet e le pagine JSP, tecnologie che permettono di ottenere una *separation of concerns* incapsulando nelle pagine JSP gli aspetti di

rendering e lasciando ai Servlet il ruolo di controller dell'applicazione web. Il problema non è comunque ancora risolto in quanto le pagine JSP contengono codice Java e si ha quindi ancora una commistione fra dettagli di visualizzazione ed elaborazione dei dati. L'utilizzo dei Cascading Style Sheet (CSS) e delle librerie di tag JSP permette di migliorare la situazione ma non rappresenta la soluzione completa.

XML e XSL entrano a questo punto nello scenario appena delineato. L'uso di XSLT, in particolare, consente di eseguire trasformazioni, per fini applicativi o di styling, su un documento. Un web designer potrebbe quindi disegnare una pagina mediante un tool che genera il corrispondente codice XSLT, senza dover modificare pagine JSP contenenti codice Java.

Si vedrà ora un esempio completo di un'applicazione di Web Publishing con Servlet, pagine JSP e XML. Nel seguito poi si descriverà Cocoon, un framework open source per il Web Publishing realizzato dall'XML Apache Group.

Il semplice esempio che si vuole realizzare deve permettere di generare pagine dinamiche mediante trasformazioni XSLT; l'architettura di base deve inoltre essere il più possibile estendibile e riutilizzabile, sfruttando le caratteristiche object oriented di Java. Si implementerà un framework, molto semplice ma tuttavia il più possibile modulare, che risponda ai requisiti dell'applicazione, ovvero deve essere possibile eseguire elaborazioni arbitrarie sui dati richiesti e fornire viste multiple sugli stessi dati a fronte di richieste diverse.

L'architettura sarà basata sul paradigma Model–View–Controller (MVC) e si comporrà dei seguenti componenti:

- controller: punto centralizzato di raccolta delle richieste e inoltro ai corrispondenti gestori
- handler: responsabile della gestione di una richiesta
- provider: fornisce i dati all'handler incapsulando i dettagli di raccolta dei dati e il formato in cui questi vengono restituiti all'handler
- data container: controlla la logica di visualizzazione dei dati. Un data container potrebbe fornire visualizzazioni diverse per gli stessi dati
- view: incapsula i dettagli di rendering dei dati

Servlet, pagine JSP e XML sono le tecnologie ideali per la realizzazione di un framework con l'architettura descritta. Le specifiche JSP presentano infatti un modello di programmazione, chiamato Model 2, basato appunto sul paradigma MVC in cui i Servlet hanno il ruolo di controller mentre le pagine JSP rappresentano la vista dei dati.

Per rendere il framework il più possibile dinamico si farà uso di un file di configurazione in cui sono specificati gli handler e le classi Java corrispondenti. Il framework prevederà

quindi solo le interfacce per gli handler e i provider; le classi specifiche verranno implementate in base alle esigenze dell'applicazione.

L'interfaccia `Handler` definisce il comportamento di un handler generico, ovvero inizializzazione, gestione di una richiesta HTTP, deallocazione. L'interfaccia definisce i seguenti metodi

```
public interface Handler {

   public void init(ServletContext context);
   public void destroy();

   public void handle(HttpServletRequest request,
                      HttpServletResponse response)throws  IOException,
                                                           ServletException;
}
```

L'interfaccia `Provider` definisce un unico metodo per reperire dati in base ai parametri specificati in una `Hashtable`:

```
public interface Provider {
   public Object getData(Hashtable parms) throws Exception;
}
```

Si osservi che il tipo del valore di ritorno di `getData()` è `Object` poiché non è possibile sapere a priori in che formato saranno restituiti i dati (documento DOM, `Resultset` JDBC, semplice file di testo, ...). L'aver incapsulato la generazione dei contenuti in un'apposita interfaccia permette poi ad esempio di riutilizzare lo stesso provider con handler diversi.

La classe più importante del framework è il Servlet che svolge il ruolo di controller ricevendo le richieste e inoltrandole agli handler corrispondenti. Tale gestione viene implementata nel metodo `handleRequest()`.

```
private void handleRequest(HttpServletRequest request,
                           HttpServletResponse response) throws IOException,
                                                                ServletException {

   String handler_name = request.getParameter("handler");
   Handler handler = getHandler(handler_name);

   if (handler != null) {
      handler.handle(request, response);
   } else {
      response.getWriter().println("Handler not found.");
   }

}
```

L'handler corrispondente a una richiesta viene specificato mediante un nome associato a un parametro, chiamato handler, nell'URL mediante la sintassi

```
http://...?handler=HandlerName
```

La funzione `getHandler()` restituisce l'oggetto corrispondente al nome indicato o `null` se l'handler non viene trovato; se la ricerca ha dato esito positivo si invoca il metodo `handle()` sull'handler. Si osservi che in questo modo la logica di elaborazione delle richieste è completamente incapsulata in oggetti applicativi che non fanno parte del framework.

Gli handler specifici di un'applicazione devono essere elencati in un file di configurazione in formato XML. Per ogni handler si dovranno indicare un nome mediante il quale identificarlo e la corrispondente classe; il seguente file mostra i dati di un handler il cui nome è `Reporter` e la cui classe è `ReportingHandler`.

```
<?xml version="1.0"?>
<handlers>
   <handler>
      <name>Reporter</name>
      <class>ReportingHandler</class>
   </handler>
</handlers>
```

All'inizializzazione del Servlet si procede alla lettura di tale file, alla creazione delle classi specificate e al loro inserimento in una tabella hash usando il nome dell'handler come chiave.

Si supponga ora di voler realizzare una semplice applicazione di reporting XML usando il framework precedente. Per semplicità i dati saranno contenuti in un file di testo; l'handler dovrà quindi ottenere i dati e inviarli a una pagina JSP per la visualizzazione mediante una trasformazione XSL. Lo stylesheet verrà determinato dall'handler in base al tipo di client (web o WAP).

Il risultato di un'elaborazione sarà quindi individuato da un documento DOM e da uno stylesheet. Rappresentare i dati mediante un documento DOM permette di astrarre dai dettagli di generazione del documento stesso; questi sono infatti incapsulati nella classe `SimpleProvider` che implementa l'interfaccia `Provider` del framework.

```
public class SimpleProvider implements Provider {

   public Object getData(Hashtable parms) throws Exception {
      String file_name;
      DOMParser parser;
      Document doc;
```

```
        parser = new DOMParser();
        file_name = (String)parms.get("datasource");
        parser.parse(file_name);
        doc = parser.getDocument();

        DOMResult result = new DOMResult();
        result.setDocument(doc);

        return result;
    }

}
```

Dopo aver recuperato il nome del file dalla hashtable si usa il parser di Xerces per caricare il file come documento DOM e restituirlo all'handler attraverso la classe `DOMResult`. Segue ora il codice dell'handler.

```
public class ReportingHandler implements Handler {

    ServletContext context;

    public ReportingHandler() { }

    public void init(ServletContext context) {
        this.context = context;
    }

    public void destroy() {
        context = null;
    }

    public void handle(HttpServletRequest request,
                       HttpServletResponse response) throws IOException,
                                                              ServletException {

        Provider provider;
        Hashtable   parms;
        DOMResult   result = null;
        String      uri;

        provider = new SimpleProvider();
        parms = new Hashtable();
        parms.put("datasource", Util.buildURI(request, "ab.xml"));
```

Una volta creato il provider si imposta l'URI corrispondente al file da reperire, costruito con la funzione di utilità `buildURI()`, che viene poi passato come parametro al metodo `getData()` con una `Hashtable`.

```
      try {
         result = (DOMResult)provider.getData(parms);
      } catch (Exception e) {
         // errore nel reperimento dei dati
         response.getWriter().println(e);
      }

      request.setAttribute("resultBean", result);
```

A questo punto si inoltra la richiesta alla pagina JSP che provvederà a eseguire la trasformazione XSL per il rendering dei dati sul client.

```
      // invia il risultato alla pagina JSP
      context.getRequestDispatcher("/report.jsp").forward(request, response);

   }
```

Il codice della pagina JSP è molto semplice; deve infatti accedere al bean passato come parametro ed eseguire una trasformazione XSL coi dati ricevuti, determinando lo stylesheet in base alle caratteristiche del client.

```
<%@ page language="java" import
="org.xml.sax.*, org.w3c.dom.*, org.apache.xalan.xslt.*, xmlweb.util.*" %>
<jsp:useBean id="resultBean" class="DOMResult" scope="request" />
<%

   // imposta lo stylesheet in base al client (HTML o WML)
   String browserType = request.getHeader("User-Agent"), stylesheet = "";

   if (browserType.indexOf("MSIE") != -1 ||
        browserType.indexOf("Mozilla") != -1 ||
        browserType.indexOf("Opera") != -1) {
      // il risultato sarà un documento HTML
      response.setContentType("text/html");
      stylesheet = Util.buildURI(request, "style-html.xsl");
   } else if (browserType.indexOf("Nokia") != -1 ||
              browserType.indexOf("UP") != -1) {
      // il risultato sarà un documento WML
      response.setContentType("text/vnd.wap.wml");
      stylesheet = Util.buildURI(request, "style-wml.xsl");
   }


   // costruisce un engine XSL Xalan
   // in grado di elaborare input DOM
   XSLTProcessor processor
   = XSLTProcessorFactory.getProcessor(new org.apache.xalan.xpath.xdom.XercesLiaison());
```

```
   // esegue la trasformazione XSL mandando il risultato al client
   processor.process(new XSLTInputSource(resultBean.getDocument()),
                                       new XSLTInputSource(stylesheet),
                                       new XSLTResultTarget(out));

%>
```

Con questo stylesheet si ottiene dal documento XML una tabella HTML.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:template match="addressbook">
      <html>
         <head><title>Adress book</title></head>
         <body>
            <table border="1">
               <th>Name</th>
               <th>Address</th>
               <th>E-mail</th>
               <th>Phone number</th>
               <xsl:apply-templates select="person"/>
               </table>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="person">
      <tr>
```

Figura 13.3 – *Documento XML trasformato in formato HTML*

```
         <td><xsl:value-of select="name"/></td>
         <td><xsl:value-of select="address"/></td>
         <td><xsl:value-of select="e-mail"/></td>
         <td><xsl:value-of select="phone"/></td>
      </tr>
   </xsl:template>
</xsl:stylesheet>
```

Il risultato della trasformazione mostrato da un browser web è illustrato in fig. 13.3.

Volendo rendere l'applicazione disponibile anche da client WAP si dovrà mostrare solo una vista parziale dei dati per far fronte alle ridotte dimensioni dei terminali. Il seguente stylesheet formatta i dati come file WML in cui compare solo l'elenco dei nomi delle persone in agenda.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output
   method="xml"
   doctype-public = "-//WAPFORUM//DTD WML 1.1//EN"
   doctype-system = "http://www.wapforum.org/DTD/wml_1.1.xml"
   media-type = "text/vnd.wap.wml"
/>


   <xsl:template match="addressbook">
      <wml>
         <card id="main" title="Adress book">
            <xsl:apply-templates select="person"/>
         </card>
      </wml>
   </xsl:template>

   <xsl:template match="person">
      <p><xsl:value-of select="name"/></p>
   </xsl:template>
</xsl:stylesheet>
```

Per maggiori dettagli su WML e WAP si rimanda a [14].

L'esempio ora presentato era volutamente molto semplificato per presentare i concetti di base del Web Publishing e mostrare come utilizzare le tecnologie basate su Java per realizzare un'infrastruttura dinamica. Si è visto come un'accurata progettazione del framework abbia permesso di separare le fasi di gestione delle richieste, generazione dei contenuti e visualizzazione, garantendo un buon grado di riuso ed estendibilità.

Alla luce dei concetti visti si descriverà ora un progetto concreto: Cocoon [16]. Cocoon è un framework per il Web Publishing realizzato in Java dall'XML Apache Group e basa-

to sulle tecnologie XML (come DOM e XSL). L'idea principale alla base di Cocoon consiste nell'individuare tre fasi nello sviluppo di un contenuto web e nella loro separazione. Le tre fasi possono essere individuate come segue:

- generazione del contenuto: questa operazione può essere eseguita da un utente attraverso un editor oppure da una generica sorgente dati (database, EJB, …).

- elaborazione: in questa fase il documento originario viene elaborato attraverso una trasformazione XSL. In questa fase si usa il termine *logicsheet* per indicare il documento XSL utilizzato. Si osservi che la logica di elaborazione del documento è separata dal documento stesso.

- rendering: il documento viene visualizzato mediante l'applicazione di uno stylesheet XSL in modo da ottenere un documento del formato richiesto, come ad esempio HTML, PDF o WML.

Il modello di publishing di Cocoon è basato su XSL che permette di collegare le fasi precedenti mediante trasformazioni XSL (XSLT). La versione al momento disponibile di Cocoon è la 1.4.7 ma sono comunque già disponibili alcune informazioni su Cocoon 2. Le note seguenti faranno riferimento a quelle caratteristiche di Cocoon che saranno mantenute anche nella versione successiva.

Cocoon si basa sulla generazione dinamica dei documenti XML da elaborare e la tecnologia chiave è rappresentata da XSP (eXtensible Server Pages). Una pagina XSP è un documento XML contenente le direttive che indicano a Cocoon come generare dinamicamente il contenuto del documento. Le direttive sono tag e lo sviluppatore può utilizzare i tag predefiniti per XSP oppure crearne di propri.

Segue ora un semplice esempio di pagina XSP, il classico Hello World leggermente modificato per mostrare la data e l'ora corrente.

```
<?xml version="1.0"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="hello-html.xsl" type="text/xsl"?>

<xsp:page language="java" xmlns:xsp="http://www.apache.org/1999/XSP/Core">
   <document>
      <xsp:logic>
         Date today = new Date();
      </xsp:logic>

      <content>Hello World, today is <xsp:expr>today</xsp:expr></content>
   </document>
</xsp:page>
```

Il documento inizia con due *processing instruction*s (d'ora in avanti:PI) che indicano all'engine di Cocoon come trasformare il documento. Il processo di elaborazione delle pagine XSP prevede che queste vengano tradotte in un programma producer; questo viene quindi compilato ed eseguito per ottenere in output il documento XML richiesto. La PI

```
<?cocoon-process type="xsp"?>
```

indica quindi a Cocoon che la pagina è XSP. L'output del producer viene poi trasformato in base allo stylesheet XSL specificato dalle successive PI

```
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="hello-html.xsl" type="text/xsl"?>
```

Seguono quindi le direttive XSP appartenenti al namespace `<xsp>`. L'elemento root del documento è `<xsp:page>` che fra i suoi attributi permette di specificare il linguaggio di programmazione usato nella pagina e il namespace XSP.

All'interno del documento la logica applicativa è incapsulata nei due tag seguenti.

```
<xsp:logic>
```

Contiene del codice che sarà ricopiato nel programma producer generato da Cocoon.

```
<xsp:expr>
```

Valuta un'espressione e inserisce un corrispondente nodo `Text` nel documento DOM risultante.

La pagina XSP precedente ha il problema di contenere al suo interno del codice che non può essere riutilizzato in altre pagine. Per risolvere questo problema è possibile definire delle librerie di tag XSP. In questo modo i dettagli relativi alla generazione dinamica dei contenuti possono essere incapsulati in tali librerie. Si supponga ad esempio di voler visualizzare i dati di una persona e di voler calcolare dinamicamente la sua età. Il calcolo dell'età verrà eseguito mediante un apposito tag, definito nel seguente documento XSL

```
<?xml version="1.0"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
   xmlns:xsp="http://www.apache.org/1999/XSP/Core"
   xmlns:services="http://www.foobar.org/DTD/XSP/Services"
>

   <xsl:template match="xsp:page">
      <xsl:processing-instruction name
      ="cocoon-process">type="xsp"</xsl:processing-instruction>
```

```
    <xsl:processing-instruction name
    ="cocoon-process">type="xslt"</xsl:processing-instruction>
    <xsl:processing-instruction name
    ="xml-stylesheet">href
    ="page-html.xsl" type
    ="text/xsl"</xsl:processing-instruction>

    <xsp:page>
       <xsl:copy>
          <xsl:apply-templates select="@*"/>
       </xsl:copy>
       <xsp:structure>
          <xsp:include>java.util.Date</xsp:include>
          <xsp:include>java.text.SimpleDateFormat</xsp:include>
       </xsp:structure>

       <xsp:logic>
          private String computeAge(String birthDate,
                                       String format) {
             Date now = new Date(),
             birth;
             int age;

             if (format == null || format.length() == 0) {
                format = "yyyy/MM/dd";
             }

             try {
                birth = new SimpleDateFormat(format).parse(birthDate);
             } catch (Exception e) {
             return "NaN";
             }
             age = now.getYear() - birth.getYear();
             birth.setYear(now.getYear());

             if (now.before(birth)) {
                --age;
             }
             return Integer.toString(age);
          }
       </xsp:logic>
       <xsl:apply-templates/>
    </xsp:page>
 </xsl:template>
       <xsl:template match="services:get-age">
             <xsp:expr>computeAge("<xsl:value-of select ='@birth-date'/>",
                         "<xsl:value-of select='@date-format'/>")</xsp:expr>
       </xsl:template>
 <xsl:template match="@*|node()" priority="-1">
```

```
      <xsl:copy><xsl:apply-templates select="@*|node()"/></xsl:copy>
   </xsl:template>
</xsl:stylesheet>
```

Lo stylesheet definisce tra gli altri un namespace separato per la libreria, chiamato in questo caso `services`. Il tag può essere quindi richiamato con

```
<services:get-age birth-date="04/25/1973" date-format="MM/dd/yyyy"/>
```

L'elemento `<xsp:structure>` include vari statement `<xsp:include>` per includere librerie esterne, in questo caso package Java.

All'interno dell'elemento `<xsp:logic>` si inseriscono le dichiarazioni visibili ai tag, come variabili e metodi. Nell'esempio si definisce il metodo `computeAge()` che accetta come parametri una data e un formato usato per costruire il corrispondente oggetto `Date`. Il metodo calcola quindi l'età di un individuo nato nella data passata come parametro, restituendo il risultato come stringa.

Al tag viene associato un template XSL che definisce quale codice generare in corrispondenza del tag in una pagina XSP. Per l'esempio visto verrà generato il codice seguente

```
<xsp:expr>computeAge("04/25/1973", "yyyy/MM/dd")</xsp:expr>
```

Segue ora una pagina XSP che utilizza il tag precedente.

```
<?xml version="1.0"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="tags-xsp.xsl" type="text/xsl"?>

<xsp:page
   language="java"
   xmlns:xsp="http://www.apache.org/1999/XSP/Core"
   xmlns:services="http://www.foobar.org/DTD/XSP/Services"
>
   <page title="XSP Demo">
      <p>Andrea is <services:get-age birth-date
      ="04/25/1973" date-format="MM/dd/yyyy"/> years old</p>
   </page>
</xsp:page>
```

Per completare il quadro segue ora il codice dello stylesheet usato per formattare il documento.

```
<?xml version="1.0"?>
```

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="page">
      <xsl:processing-instruction name="cocoon-format">
         type="text/html"
      </xsl:processing-instruction>
      <html>
         <head><title><xsl:value-of select="title"/></title></head>
         <body>
            <xsl:apply-templates/>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="p">
      <xsl:copy>
         <xsl:apply-templates/>
      </xsl:copy>
   </xsl:template>
</xsl:stylesheet>
```

Come visto dall'esempio la tecnologia XSP permette di ottenere un'architettura in cui contenuti, logica applicativa e presentazione sono completamente separati.

Cocoon è uno strumento molto interessante anche se al momento ancora immaturo. Un grosso problema, comune a tutti i tool di publishing basati su XSL, è dato dalla scarsa disponibilità di editor XSL visuali. Un approccio basato su pagine JSP potrebbe limitare il problema, permettendo al web designer di progettare lo scheletro della pagina mediante i tool disponibili; il programmatore provvederebbe a popolare la pagina con documenti XML trasformati via XSL. In questo caso la maggior parte dei dettagli grafici sarebbe incapsulata nella pagina JSP permettendo al programmatore di concentrarsi solo sulla logica applicativa.

Integrazione di applicazioni

Come si è visto in precedenza, XML rappresenta un formato standard per la rappresentazione dei dati e questo è un grosso vantaggio per l'integrazione di applicazioni. A questo punto è necessario trovare un meccanismo di comunicazione distribuito che fornisca un valore aggiunto nello sviluppo di transazioni con dati rappresentati in XML. Si possono distinguere due meccanismi di comunicazione:

- sincrono, come ad esempio RMI

- asincrono, basato sullo scambio di messaggi mediante un mezzo di comunicazione/coordinazione condiviso

In questo paragrafo si analizzeranno due implementazioni Java based di tali infrastrutture di comunicazione, SOAP e JMS. SOAP, come si vedrà tra poco, è un protocollo che sfrutta XML come formato di rappresentazione dei dati. JMS è invece una specifica per l'implementazione di servizi di messaging asincrono in Java che ben si presta allo scambio di documenti XML.

SOAP

SOAP (Simple Object Access Protocol) [17] è un protocollo di interazione fra componenti remoti basato su XML e HTTP. Il protocollo si compone di tre parti:

- un envelope che rappresenta un framework per descrivere il contenuto di un messaggio e il modo in cui elaborarlo (SOAP envelope)
- una serie di regole di codifica per rappresentare i tipi di dato definiti per l'applicazione (SOAP encoding rules)
- delle convenzioni per rappresentare le invocazioni remote e le corrispondenti reply (SOAP RPC)

In pratica si tratta di un meccanismo di RPC in cui le richieste e le reply vengono definite mediante XML mentre il protocollo di trasporto è HTTP. L'utilizzo di XML per la descrizione di servizi distribuiti permette la massima interoperabilità.

Il protocollo vuole essere semplice ed estendibile quindi le specifiche non includono caratteristiche come garbage collection distribuita e gestione dei riferimenti remoti, tipiche dei sistemi a oggetti distribuiti.

SOAP si appoggia pesantemente sull'architettura web in quanto sfrutta HTTP come protocollo di trasporto. Altri meccanismi di computazione distribuita, come DCOM, Java RMI e CORBA, si integrano con difficoltà su web in quanto, ad esempio, possono essere bloccati dai firewall.

SOAP è una proposta di varie aziende, fra le quali Microsoft, IBM e DevelopMentor, sottoposta al W3C e all'IETF per la standardizzazione. Il protocollo è alla base dell'architettura .NET di Microsoft e di Web Services di IBM. L'XML Apache Group fornisce un'implementazione Java, chiamata Apache–SOAP, basata sulla versione 1.1 del protocollo (`http://xml.apache.org/soap`); l'esempio presentato in questa sezione farà uso di Apache–SOAP.

Un messaggio SOAP è un documento XML costituito da un envelope, un header e un body. Il SOAP envelope è obbligatorio e rappresenta il messaggio. L'header è un elemento opzionale che contiene informazioni che estendono il messaggio, relative ad esempio a gestione delle transazioni e autenticazione. Il body è un elemento obbligatorio che per-

mette di trasmettere informazioni destinate al ricevente del messaggio. Un messaggio SOAP può infatti transitare attraverso diversi nodi intermedi. Eventuali errori vengono rappresentati dall'elemento SOAP `Fault`.

In SOAP sono previsti due tipi di messaggi, Call e Response. Un messaggio di tipo Call permette di invocare un servizio remoto. Si supponga ad esempio di voler richiedere la somma di due numeri a un server remoto; la richiesta in SOAP verrebbe espressa con il seguente messaggio.

```
POST /Calculator HTTP/1.1
Host: www.mycalculator.com
Content-Type: text/xml
Content-Length: x
SOAPMethodName: My_Namespace_URI#GetSum

<SOAP:Envelope xmlns:SOAP=http://schemas.xmlsoap.org/soap/envelope"">
   <SOAP:Body>
      <m:GetSum xmlns="http://www.mycalculator.com">
         <first>3</first>
         <second>8</second>
      </m:GetSum>
   </SOAP:Body>
</SOAP:Envelope>
```

Le prime quattro linee sono specifiche del protocollo HTTP e indicano che si tratta di una richiesta POST inviata all'host `http://www.mycalculator.com` per il servizio `Calculator`. Il messaggio è contenuto all'interno della SOAP envelope: l'elemento `GetSum` contiene i parametri forniti per il servizio, in questo caso due numeri da sommare individuati dagli elementi `first` e `second`.

I messaggi Response contengono il risultato dell'elaborazione del servizio. Il messaggio Response corrispondente alla richiesta precedente sarebbe

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: x

<SOAP:Envelope xmlns:SOAP=http://schemas.xmlsoap.org/soap/envelope"">
   <SOAP:Body>
      <m:GetSumResponse xmlns="http://www.mycalculator.com">
         <return>11</return>
      </m:GetSumResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

In questo caso la SOAP envelope contiene il risultato dell'elaborazione come valore dell'elemento `return`.

Il modello a oggetti di Apache–SOAP viene esposto dal package `org.apache.soap.rpc`. Il client che vuole usare un servizio remoto utilizza l'oggetto `Call` mediante il quale specifica il nome del metodo, l'ID e i parametri. Una volta impostato l'oggetto `Call` è possibile invocarne il metodo

```
invoke(URL url, String SOAPActionURI)
```

nei cui parametri si indicano rispettivamente l'URL in cui si trova il componente che fornisce il servizio e l'header della richiesta.

Il metodo `invoke()` restituisce un oggetto `Response` che contiene la risposta al servizio richiesto oppure una segnalazione di errore.

Apache–SOAP supporta i protocolli HTTP e SMTP. Se usato su HTTP si ha il seguente processo di funzionamento

- il client esegue un POST di un SOAP envelope al server;
- questo a sua volta costruisce un oggetto `Call`, localizza l'oggetto destinatario basandosi sull'object ID, verifica il nome del metodo e quindi chiama il metodo `invoke()` sull'oggetto;
- il metodo `invoke()` restituisce un oggetto `Response` da cui il server estrae le informazioni da inserire nel messaggio Response da inviare al client.

Si vedrà ora un esempio di servizio esposto via SOAP. Si realizzerà in Java un semplice servizio di somma remota come nell'esempio precedente. Sia il client che il server verranno implementati in Java per ragioni "didattiche". Si noti che potrebbero essere realizzati in qualsiasi linguaggio per il quale esista una implementazione di SOAP, come ad esempio Perl o linguaggi COM-enabled in ambiente Microsoft.

Segue ora il codice, molto semplice, del server che esegue la somma.

```
public class MyCalculator {

   public static int getSum(int arg1, int arg2) {
      return (arg1 + arg2);
   }

}
```

Per fare in modo che il servizio precedente sia visibile via Apache–SOAP è necessario scrivere un deployment descriptor in XML per la classe precedente.

```
<?xml version="1.0"?>
```

```
<n:service xmlns:n="http://xml.apache.org/xml-soap/deployment"
id="urn:xml-soap-demo-mycalculator">
   <n:provider type="java"
         scope="Application"
         methods="getSum">
      <n:java class="MyCalculator" static="true"/>
   </n:provider>
</n:service>
```

Una volta installato Apache–SOAP è possibile eseguire il deploy del servizio. Questo può essere fatto attrraverso un'apposita pagina web oppure da linea di comando.

Il seguente codice mostra come impostare l'invocazione del servizio di somma. Per semplicità è stata tralasciata qualsiasi gestione degli errori.

```
Call call = new Call();
call.setTargetObjectURI("urn:xml-soap-demo-mycalculator");
call.setMethodName("getSum");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

Vector parms = new Vector();
parms.addElement(new Parameter("first", int.class, 3, null));
parms.addElement(new Parameter("second", int.class, 8, null));

call.setParams(parms);
URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");
Response response = call.invoke(url, "");

Parameter result = response.getReturnValue();
System.out.println("Il risultato è " + ((Integer)result.getValue()).intValue());
```

SOAP è un passo molto importante verso l'integrazione di sistemi eterogenei e offre i grandi vantaggi della semplicità: non esiste infatti una API alla quale debbano conformarsi le implementazioni del protocollo. L'implementazione in Java fornita dall'Apache Group permette di rendere i programmi Java interoperabili via SOAP e si è dimostrata molto semplice da usare.

JMS

L'approccio alla comunicazione basato su RPC mostra dei limiti in un ambiente distribuito su larga scala; si pensi ad esempio a un sistema di raccolta ordini: esso dovrà essere in grado di accettare richieste da programmi diversi in tempi diversi. Un sistema basato su comunicazione sincrona dovrebbe far fronte a problemi di latenza della rete e a potenziali guasti delle stazioni di inserimento degli ordini. Per sistemi distribuiti i cui componenti siano ad accoppiamento lasco risulta particolarmente adatto un sistema di messaging asincrono che, rispetto ai sistemi RPC, presenta diversi vantaggi:

- la comunicazione asincrona non richiede che le entità comunicanti siano attive nello stesso intervallo di tempo;

- la latenza nella comunicazione non rappresenta più un problema.

Java Messaging Service (JMS) è una specifica per una API che accede a servizi di messaging, così come JDBC è una API per l'accesso a database.

Nella terminologia JMS l'implementazione delle specifiche prende il nome di JMS provider. L'architettura tipica è basata su un broker di messaggi al quale si collegano vari client. Un client individua un broker via JNDI e quindi stabilisce una connessione con esso. Durante il periodo di validità di una connessione il client crea una o più sessioni caratterizzate da diversi comportamenti transazionali e di acknowledgement.

I modelli di messaging supportati sono due: *publish/subscribe* e *point-to-point*.

Nel messaging publish/subscribe si individuano un'entità publisher che produce messaggi, e i subscriber che li ricevono, tutti nel contesto di un particolare topic. Nel modello di programmazione point-to-point vengono invece definite delle code attraverso le quali vengono scambiati messaggi gestiti in maniera FIFO. Il broker mantiene cioè un insieme di messaggi fin quando il messaggio corrente non è stato consumato.

I messaggi JMS vengono rappresentati dall'interfaccia `Message` e sono composti da tre sezioni:

- header: informazioni usate dai client e dai provider per la gestione dei messaggi

- proprietà: ad ogni messaggio possono essere associate delle proprietà definite dall'utente e utilizzabili per una migliore granularità nel filtro dei messaggi

- body: il contenuto del messaggio. JMS supporta cinque tipi di messaggio, a cui corrispondono altrettante interfacce che estendono `Message`. Esse sono:

  `BytesMessage`

  Uno stream di byte.

  `MapMessage`

  Un insieme di coppie name–value dove name è un oggetto `String` mentre value è un tipo primitivo Java. L'accesso può essere sequenziale o associativo.

  `ObjectMessage`

  Un oggetto Java serializzato.

`StreamMessage`

Una sequenza di oggetti primitivi Java, scritta e letta in modo sequenziale.

`TextMessage`

Messaggio contenente un oggetto `StringBuffer` ovvero un potenziale documento XML.

Per maggiori approfondimenti sulla tecnologia JMS si veda [18].

Integrando messaggi XML a JMS si ottiene un'infrastruttura indipendente dalla piattaforma e dal formato dati utilizzati. Al momento le specifiche JMS non prevedono alcun supporto diretto per i documenti XML. Esistono comunque alcuni JMS provider che forniscono una qualche forma di supporto a XML come SonicMQ di Progress.

Si vedrà ora un esempio di applicazione di messaging XML; si vuole realizzare una classe Java in grado di inviare e ricevere messaggi JMS in formato XML, in modo che possa fare da tramite fra applicazioni XML based e sistemi di messaging. I messaggi XML saranno inglobati in oggetti `TextMessage`, gestiti dalla classe `XMLAgent`.

```
public class XMLAgent {
   ...
```

Il metodo `sendXMLMessage()` accetta come parametro una stringa contenente un documento XML e provvede a costruire un messaggio JMS e a inviarlo secondo i parametri impostati nel costruttore:

```
   public void sendXMLMessage(String xmlMessage) throws JMSException,
                                                        SAXException,
                                                        IOException {
      TopicPublisher publisher = session.createPublisher(topic);
      TextMessage message = session.createTextMessage(xmlMessage);

      connection.start();
      XMLUtils.parseXMLDocument(xmlMessage);
      publisher.publish(message);
      connection.close();
   }
```

La ricezione dei messaggi è svolta dal metodo `listenXMLMessage()` che accetta come parametro un oggetto handler: per rendere generico il sistema si definisce infatti un'interfaccia callback (`MessageHandler`) che permette di ricevere la notifica della ricezione di un messaggio, preventivamente trasformato in documento DOM.

```
   public void listenXMLMessage(MessageHandler handler) throws JMSException,
                                                              SAXException,
                                                              IOException {
```

```
      TopicSubscriber subscriber
      = session.createSubscriber(topic, null, false);
      Message message;
      Document document;

      connection.start();
      message = subscriber.receive();
      connection.close();

      if (message instanceof TextMessage) {
         document = XMLUtils.getDOMDocument(((TextMessage)message).getText());
         handler.handle(document);
      }
   }

}
```

La classe `XMLUtils` utilizzata dai metodi precedenti incapsula la gestione del parsing dei documenti XML, nascondendo ogni dettaglio sul particolare parser utilizzato.

La sinergia fra XML e sistemi di messaging offre un grande potenziale per l'integrazione di applicazioni e molti prodotti sono basati su queste tecnologie come BizTalk Server di Microsoft, Fortè Fusion e MQSeries Integrator (gli ultimi dei quali basati su Java). In questa sezione si è approfondito l'argomento e si è visto come la piattaforma Java fornisca tutti gli strumenti per affrontare il problema dell'integrazione.

Conclusioni

XML rappresenta senza dubbio un must per ogni programmatore e in questo capitolo si è visto come siano disponibili vari tool in Java che permettono di sfruttare appieno le tecnologie XML e come le tecnologie Java (Servlet, JSP, JMS, …) si integrino con XML per costituire una solida infrastruttura per applicazioni distribuite, principalmente Web Publishing e integrazione di applicazioni.

L'argomento Java e XML non si completa sicuramente con questo capitolo. Per completare il quadro sarebbe opportuna una rassegna dei tool Java (commerciali e open source) disponibili, in modo anche da valutare nuovi utilizzi di XML. Ad esempio KaanBaan [8] propone l'integrazione di XML con gli Enterprise Java Bean per lo sviluppo di applicazioni transazionali mentre il progetto Xbeans è basato sull'utilizzo di Java Bean per l'elaborazione di documenti XML in applicazioni distribuite [21]. Quanto visto permette comunque di avere solide basi per approfondire la programmazione di applicazioni XML based in Java.

Bibliografia

[1] Specifiche W3C di XML: `http://www.w3c.org/XML`

[2] Schema XML: `http://www.w3.org/XML/Schema.html`

[3] Il sito di SAX: `http://www.megginson.com/SAX`

[4] Il sito del W3C dedicato a DOM: `http://www.w3c.org/DOM`

[5] Il sito del progetto JDOM: `http://www.jdom.org`

[6] Sun Data Binding Specification Request:
`http://java.sun.com/aboutJava/communityprocess/jsrjsr_031_xmld.html`

[7] Enhydra, un application server open source basato su Java e XML: `http://www.enhydra.org`

[8] KaanBaan, un transaction server che integra XML e la tecnologia EJB 2.0: `http://www.kaanbaan.com`

[9] Breeze XML Studio: `http://www.breezefactor.com/`

[10] XSL: `http://www.w3.org/TR/xsl`

[11] Apache XML FOP (Formatting Objects Project): `http://xml.apache.org/fop`

[12] XPath: `http://www.w3.org/TR/xpath`

[13] André Tost, *XML document processing in Java using XPath and XSLT*, "JavaWorld", settembre 2000; `http://www.javaworld.com/jw-09-2000/jw-0908-xpath.html`

[14] Il WAP forum: `http://www.wapforum.org`

[15] P.SDK di Phone.com: `http://www.phone.com`

[16] Cocoon: `http://xml.apache.org/cocoon`

[17] Le specifiche su SOAP del W3C: `http://www.w3c.org/TR/soap`

[18] GORDON VAN HUIZEN, *JMS: An infrastructure for XML-based business-to-business communication*, "JavaWorld", febbraio 2000; `http://www.javaworld.com/jw-02-2000/jw-02-jmsxml.html`

[19] Sito della Sun dedicato a Java e XML: `http://java.sun.com/xml`

[20] Sito di IBM che fornisce vari articoli, tool e informazioni su XML con riferimento a Java: `http://www.ibm.com/developer/xml`

[21] Xbeans, progetto open source basato sull'utilizzo di Java Beans per l'elaborazione di documenti XML in applicazioni distribuite: `http://www.xbeans.org`

Capitolo 14

Servlet API

DI GIOVANNI PULITI

Come noto Java è un linguaggio fortemente orientato alla programmazione delle reti e a quella distribuita.

Grazie alla facilità con cui questo linguaggio permette ad esempio di gestire le comunicazioni via socket e oggetti remoti per mezzo di RMI, è possibile realizzare applicazioni multistrato e distribuite in modo relativamente semplice.

Uno dei settori sul quale si focalizza maggiormente l'attenzione dello scenario della Information Technology è quello della programmazione web oriented, ovvero quella in cui la parte client è costituita da un semplice browser che interagisce con la parte server per mezzo del protocollo HTTP.

Questa tipologia di programmi, il cui modello di funzionamento viene tipicamente denominato Common Gateway Interface (CGI), ha come obiettivo quello di permettere l'interfacciamento da parte di un client web con una serie di risorse e di servizi residenti sul server.

Sebbene CGI sia una definizione generica, con tale sigla si fa riferimento in genere a programmi scritti utilizzando linguaggi come il C o il PERL, ovvero secondo una filosofia di progettazione ormai superata.

Dal punto di vista della logica di esecuzione, questa è la sequenza delle operazioni:

- il client tramite browser effettua una chiamata al web server;
- il web server provvede a eseguire l'applicazione;
- l'applicazione dopo aver eseguito tutte le operazioni del caso, produce un output che viene passato al web server che lo invierà poi al client.

Poco prima dell'uscita definitiva del JDK 1.2, per la realizzazione di applicazioni CGI in Java Sun ha introdotto la Servlet API, diventata in poco tempo una delle più importanti di tutta la tecnologia Java.

I Servlet hanno introdotto alcune importanti innovazioni nel modello operativo del CGI, innovazioni che poi sono state adottate di riflesso sia nelle tecnologie più obsolete, sia in quelle direttamente concorrenti.

Nella terminologia dell'informatica si presenta spesso il problema della scelta del genere (maschile o femminile) per i termini che in inglese sono di genere neutro. Alcuni autori propongono di utilizzare il genere della "traduzione ideale" del termine in lingua italiana. Nel caso di "Servlet", per esempio, si potrebbe utilizzare la forma femminile, dato che "una" Servlet è in definitiva "una applicazione lato server". In realtà, spesso si segue tutt'altra strada e il genere corretto finisce per essere quello invalso nell'uso comune. La disputa è quanto mai accesa e anche in questo capitolo verranno adottate varie forme, pur privilegiando il maschile. Chi si dovesse trovare più a suo agio con il femminile sappia che non commette errore e può contare su un ampio numero di "colleghi".

La Servlet API

Secondo la definizione ufficiale, un Servlet è "componente lato server per l'estensione di web server Java enabled", cioè un qualcosa di più generale del solo WWW. Un Servlet è quindi un programma Java in esecuzione sul server ed in grado di colloquiare con il client per mezzo del protocollo HTTP. Tipicamente questo si traduce nella possibilità di generare dinamicamente contenuti web da visualizzare nella finestra del client-browser.

I packages che contengono tutte le classi necessarie per la programmazione dei Servlet sono il `javax.servlet` ed il `javax.servlet.http`: come si può intuire dal loro nome si tratta di Standard Extension API, ovvero non fanno parte del JDK core.

Per questo motivo, quando si lavora con i Servlet, è importante specificare quale versione della API si utilizza. Infatti, dato che il Servlet engine utilizzato potrebbe non supportare l'ultima versione delle API, spesso si rende necessario utilizzare una versione più vecchia dell'ultima disponibile.

Quindi, dove necessario, verranno affrontati i vari aspetti delle API indicando specificatamente ove siano presenti delle differenze fra le varie API: le versioni a cui si fa riferimento in questa sede saranno la 2.0, la 2.1 e la 2.2, mentre si tralasceranno i dettagli relativi alla 1.0, trattandosi di una versione piuttosto vecchia, non più supportata dalla maggior parte dei server.

Si tenga presente inoltre che con la versione 2.2, le Servlet API sono entrate a far parte formalmente della cosiddetta Java 2 Enterprise Edition, al fianco di altre importanti tecnologie come JDBC, JNDI, JSP, EJB, e RMI.

Questa nuova organizzazione logica, anche se non ha alterato l'architettura di base dei Servlet, ha introdotto alcune nuove definizioni e formalismi. L'innovazione più importante da questo punto di vista è l'introduzione del concetto di container al posto di server: un container è un oggetto all'interno del quale il Servlet vive e nel quale trova un contesto di esecuzione personalizzato. Il concetto di container ha poi una visione più ampia dato che viene adottato anche nel caso di JSP ed EJB. In questo capitolo si utilizzerà genericamente il termine "server" per indicare il motore d'esecuzione del Servlet stesso, motore che a seconda dei casi potrà essere un application server in esecuzione sopra i vari web server disponibili, oppure potrà essere un web server che integri una JVM al suo interno.

Soluzione full java

Essendo composto al 100% da classi Java, un Servlet è integrabile con il resto della tecnologia Java dando luogo a soluzioni scalabili in maniera molto semplice.

Efficienza

Sebbene per quanto riguarda l'invocazione il meccanismo sia piuttosto simile al CGI, dal punto di vista del funzionamento si ha una situazione radicalmente diversa. Infatti ogni volta che il client esegue una richiesta di un servizio basato su CGI, il web server deve mandare in esecuzione un processo dedicato per quel client. Se *n* client effettuano la stessa richiesta, allora *n* processi devono essere prima istanziati ed eseguiti poi. Questo comporta un notevole dispendio di tempo e di risorse di macchina. Un Servlet invece, una volta mandato in esecuzione (cioè inizializzato), può servire un numero *n* di richieste senza la necessità di ulteriori esecuzioni. In questo caso infatti il server manda in esecuzione una sola JVM, la quale effettua un caricamento dinamico delle classi necessarie per eseguire i vari Servlet invocati. Il Servlet in questo contesto prende forma di servizio. Tutto questo porta a notevoli vantaggi dal punto di vista della efficienza e delle potenzialità operative.

Migliore interfacciamento

La separazione fra i metodi `init` e `service` permette non solo di ottimizzare le risorse ma anche di migliorare la modalità di interfacciamento con il client. Il primo infatti serve per inizializzare il Servlet stesso, ed è il posto dove tipicamente si eseguono le operazioni computazionalmente costose da effettuare una volta per tutte (come la connessione a un DB). Il metodo `service` invece è quello che viene mandato in esecuzione al momen-

to di una chiamata `POST` o `GET`. Il server da questo punto di vista offre una gestione tramite thread del metodo `service()`, ovvero per ogni client che effettui una richiesta di servizio, server manderà in esecuzione un thread che eseguirà il metodo `service` (o, in maniera equivalente, il `doGet()` o `doPost()` a seconda della implemtenzione). Il tutto in modo automatico e trasparente agli occhi del programmatore che dovrà solo preoccuparsi di scrivere il codice relativo alle operazioni da effettuare in funzione della richiesta di *un* client. Eventualmente si tenga presente che è possibile realizzare Servlet monothread, in grado di servire un client per volta.

Leggerezza

I due punti precedenti fanno capire come in realtà, oltre a una notevole semplicità di programmazione, il grosso vantaggio risiede nella leggerezza con cui il tutto viene gestito.

Portabilità

Infine essendo Java il linguaggio utilizzato per lo sviluppo di Servlet, è relativamente semplice pensare di portare da un web server a un altro tutto il codice scritto senza la necessità di complesse conversioni. Tipicamente questa è una procedura molto utilizzata per passare dalla fase di sviluppo a quella operativa.

Un Servlet una volta correttamente compilato e installato nel server, segue un suo ciclo di vita ben preciso, composto dalla inizializzazione, dalla gestione delle invocazioni da parte dei client, e dalla conclusiva disintallazione. Ognuna di queste fasi è particolarmente importante, dato che condiziona sia la logica di funzionamento complessiva, sia le performance dell'applicazione nel suo complesso.

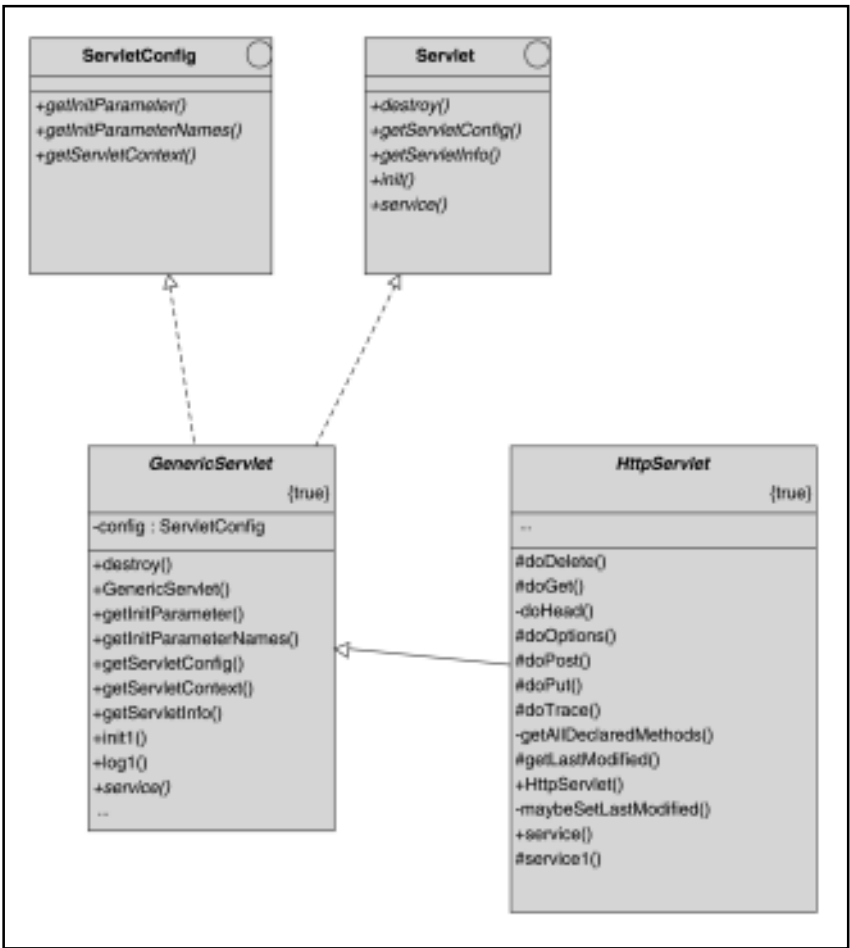
Implementazione e ciclo di vita di un Servlet

Per realizzare un Servlet HTTP è sufficiente estendere la classe `HttpServlet`, appartenente al package `javax.servlet.http`, ridefinire il metodo `init()`, per personalizzare l'inizializzazione del Servlet, ed i metodi `service()`, `doPost()` e `doGet()` per definire invece il comportamento del Servlet in funzione delle invocazioni del client.

Inizializzazione di un Servlet

Il metodo `init()`, derivato dalla classe `GenericServlet`, ha lo scopo di effettuare tutte quelle operazioni necessarie per l'inizializzazione del Servlet stesso, e per il corretto funzionamento successivo. La firma del metodo è la seguente

```
public void init(ServletConfig config) throws ServletException
```

Figura 14.1 – *Gerarchia delle classi principali per la gestione dei Servlet*

Se durante il processo di inizializzazione del Servlet si verifica un errore tale da compromettere il corretto funzionamento successivo, allora il Servlet potrà segnalare tale evento generando una eccezione di tipo `ServletException`.

In questo caso il Servlet non verrà reso disponibile per l'invocazione, e l'istanza appena creata verrà immediatamente rilasciata. Il metodo `destroy()` in questo caso non verrà invocato dal server, considerando il Servlet non correttamente inizializzato. Dopo il rilascio dell'istanza fallita, il server procederà immediatamente, alla istanziazione ed inizializzazione di un nuovo Servlet; la generazione di una `UnavailableException` permette di specificare il tempo minimo necessario da attendere prima di intraprendere

nuovamente il processo di inizializzazione. Il costruttore da utilizzare per creare questo tipo di eccezione è il seguente

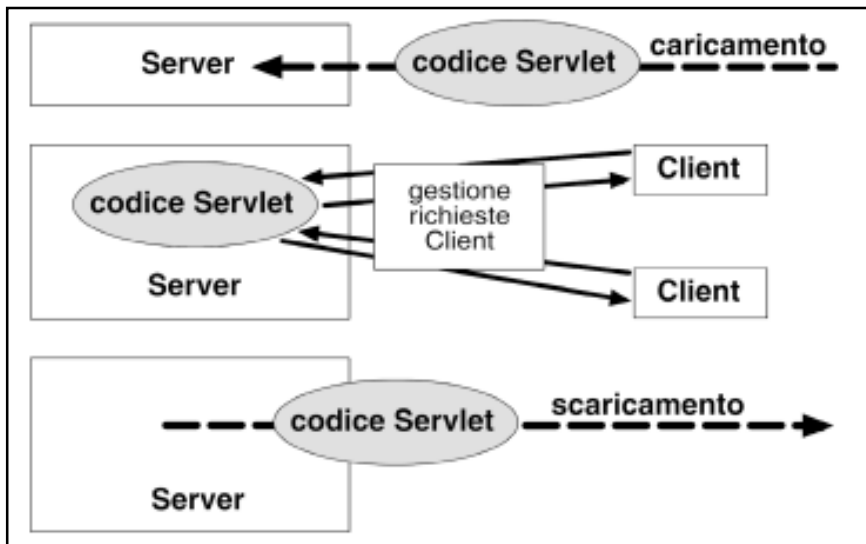
```
public UnavailableException(java.lang.String msg, int seconds)
```

dove `seconds` è il tempo minimo di attesa. Un valore negativo o pari a zero indica il caso in cui sia impossibile effettuare delle stime precise: in tal caso il server istanzierà il Servlet appena possibile.

Questo modo di gestire le anomalie lascia il controllo al Servlet container, e permette quindi di implementare strategie di backup: si deve perciò evitare l'invocazione del metodo `System.exit()`.

Un caso tipico in cui può verificarsi il fallimento della inizializzazione di un Servlet può essere quello in cui durante la `init()` si effettui una connessione con un database, utilizzata in seguito per inviare dati verso un client sotto forma di tabelle HTML. In questo caso potrebbe essere particolarmente utile imporre un periodo di attesa prima di effettuare una nuova connessione, ovvero prima di procedere alla creazione di un nuovo Servlet.

Uno dei fatti di cui tener conto, quando si utilizza la versione 2.0 della API, è che, ridefinendo il metodo `init()`, la prima cosa da fare è invocare lo stesso metodo della classe padre.

Figura 14.2 – *Ciclo di vita di un Servlet. Si noti come per ogni client venga eseguito un thread separato*

Ad esempio

```
public class myServlet {
...
   public void init(ServletConfig config) throws ServletException {
      super.init(config);
   }
   ...
}
```

può apparire una operazione inutile, ma risulta essere particolarmente importante per permettere alla `GenericServlet` di effettuare altre operazioni di inizializzazione utilizzando il riferimento alla interfaccia `ServletConfig` passata come parametro.

Con la versione 2.1 della API questa operazione non è più necessaria ed è possibile ridefinire direttamente il metodo `init()` senza parametri; è quindi possibile scrivere

```
public void init() throws ServletException {
   String UserId = getInitParameter("uid");
}
```

In questo caso il riferimento alla `ServletConfig` non viene perso dato che la classe `GenericServlet` permette questa apparente semplificazione. Il server infatti, al momento della inizializzazione del Servlet, effettua sempre una chiamata al metodo `init(ServletConfig config)`, ma la differenza è che con la versione 2.1 delle API il `GenericServlet` effettua immediatamente la chiamata alla `init` senza parametri. Ad esempio

```
public class GenericServlet implements Servlet, ServletConfig {

   ServletConfig Config = null;

   public void init(ServletConfig config) throws ServletException {
      Config = config;
      log("init called");
      init();
   }

   public void init()throws ServletException {
   }

   public String getInitParameter(String name) {
      return Config.getInitParameter(name);
   }

   // ecc...
}
```

Grazie al metodo `getInitParameter(String)`, il quale effettua una invocazione

direttamente al runtime su cui è in esecuzione il Servlet, all'interno del metodo `init()` è possibile ricavare il valore dei parametri di inizializzazione.

Il metodo `getParameterNames()` permette di conoscere il nome dei parametri di inizializzazione.

Il resto della vita: i metodi `doGet()`, `doPost()` e `service()`

Dopo l'inzializzazione, un Servlet si mette in attesa di una eventuale chiamata da parte del client, che potrà essere indistintamente una `GET` o una `POST` HTTP.

L'interfaccia `Servlet` mette a disposizione a questo scopo il metodo

```
public void service(HttpServletRequest req,
                    HttpServletResponse res) throws  ServletException,
                                                     IOException {
```

che viene invocato direttamente dal server in modalità multithread (il server per ogni invocazione da parte del client manda in esecuzione un metodo service in un thread separato).

Il `service` viene invocato indistintamente sia nel caso di una invocazione tipo `GET` che di una `POST`. La ridefinizione del metodo `service` permette di definire il comportamento del Servlet stesso. Se nel Servlet è definito il metodo `service()`, esso eseguito al posto dei metodi `doGet()` e `doPost()`.

I due parametri passati sono `ServletRequest` e `ServletResponse`, che permettono di interagire con la richiesta effettuata dal client e di inviare risposta tramite pacchetti HTTP. In seguito saranno analizzati in dettaglio gli aspetti legati a queste due interfacce.

Nel caso in cui invece si desideri implementare un controllo più fine si possono utilizzare i due metodi `doGet()` e `doPost()`, che risponderanno rispettivamente ad una chiamata di tipo `GET` e a una di tipo `POST`.

Le firme dei metodi, molto simili a quella del metodo `service()`, sono:

```
protected void doGet(HttpServletRequest req,
                    HttpServletResponse resp) throws ServletException,
                                                     IOException


protected void doPost(HttpServletRequest req,
                    HttpServletResponse resp) throws  ServletException,
                                                     IOException
```

Con la API 2.2 è stato aggiunto all'interfaccia `ServletConfig` il metodo `getServletName()` che restituisce il nome con cui il Servlet è stato registrato, o il nome della classe nel caso di un Servlet non registrato.

Il codice che segue mostra come ricavare il nome del Servlet per poter reperire ulteriori informazioni dal contesto corrente.

```
public void service(HttpServletRequest req,
                    HttpServletResponse res) throws  ServletException,
                                                     IOException {
   // invoca il metodo getServletName derivato
   // dalla classe padre GenericServlet
   String name = getServletName();
   // ricava il contesto
   ServletContext context = getServletContext();
   // ricava un attributo generico del servlet
   Object value = context.getAttribute(name + ".state");
}
```

La vita è piena di imprevisti: la gestione delle eccezioni

La `ServletException` rappresenta l'eccezione tipicamente generata in tutti quei casi in cui uno dei metodi legati al ciclo di vita del Servlet fallisca per qualche motivo. In base alla filosofia adottata nella gestione delle eccezioni si può quindi dire che la generica `ServletException` funziona come wrapper della particolare eccezione che di volta in volta verrà generata.

A volte una eccezione di questo tipo viene detta causa radice o, con un termine dal suono più familiare, *root cause*.

A questo scopo sono stati introdotti due nuovi costruttori della `ServletException` in grado di offrire una maggiore potenza descrittiva circa le cause dell'eventuale problema.

```
ServletException(Throwable rootCause)

ServletException(String message, Throwable rootCause)
```

Si supponga ad esempio di dover intercettare una `InterruptedException` e di propagare quindi una generica `ServletException`: utilizzando le Servlet API 2.0 si potrebbe scrivere

```
try {
   thread.sleep(100);
}
catch (InterruptedException e) {
   // si genera una eccezione generica
   // utilizzando esclusivamente il messaggio
   // testuale per specificare ulteriori dettagli
   throw new ServletException(e.getMessage());
}
```

che con la versione 2.1 potrebbe diventare

```
try {
```

```
   thread.sleep(100);
}
catch (InterruptedException e) {
   // si genera una eccezione specifica
   throw new ServletException(e);
}
```

In questo caso, dato che si propaga l'eccezione utilizzando un oggetto per la creazione della `ServletException`, il server riceve molte più informazioni rispetto a prima, dato che si utilizza un oggetto vero e proprio al posto di un semplice messaggio.

Questa nuova possibilità messa a disposizione dalle API 2.1 in effetti è utile, anche se un suo utilizzo comporta il rischio di dover riscrivere tutta la gestione delle eccezioni all'interno della stessa applicazione.

Per questo motivo, visto che l'aiuto fornito non è di importanza essenziale, si consiglia spesso di farne a meno.

Come interagire con i Servlet: richieste e risposte

Un Servlet può comunicare con il client in maniera bidirezionale per mezzo delle due interfacce `HttpServletRequest` e `HttpServletResponse`: la prima rappresenta la richiesta e contiene i dati provenienti dal client, mentre la seconda rappresenta la risposta e permette di incapsulare tutto ciò che deve essere inviato indietro al client stesso.

Domandare è lecito: `HttpServletRequest`

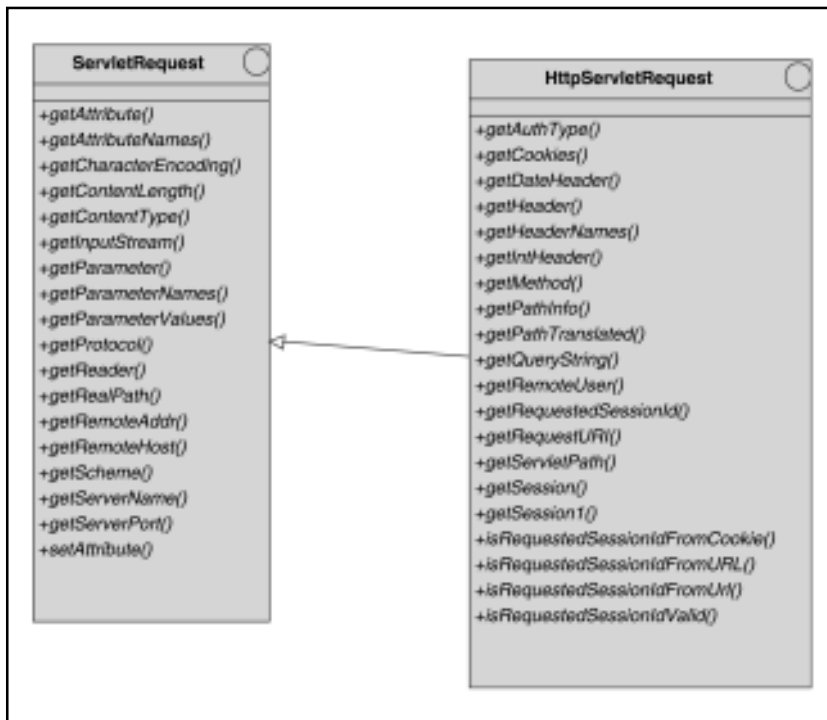
La `HttpServletRequest`, oltre a permettere l'accesso tutte le informazioni relative allo header `http` (come ad esempio i vari cookies memorizzati nella cache del browser), permette di ricavare i parametri passati insieme all'invocazione del client.

Tali parametri sono inviati come di coppie nome–valore, sia che la richiesta sia di tipo `GET` che `POST`. Ogni parametro può assumere valori multipli.

L'interfaccia `ServletRequest` dalla quale discende la `HttpServletRequest` mette a disposizione alcuni metodi per ottenere i valori dei parametri passati al Servlet: ad esempio, per ottenere il valore di un parametro dato il nome, si può utilizzare il metodo fornito dalla interfaccia

```
public String getParameter(String name)
```

Nel caso di valori multipli per la stessa variabile, come quando l'invocazione viene fatta tramite un form HTML, fino alla API 2.0 non era specificato il formalismo con cui tali parametri dovevano essere restituiti dal server, il quale poteva restituire un array, una lista di stringhe separate da virgola o altro carattere.

Figura 14.3 – *Gerarchia delle classi* `ServletRequest` *e* `HttpServletRequest`

Con la API 2.1 invece si ottiene sempre un array di stringhe, risultato analogo a quello fornito dal metodo

```
public String[] getParameterValues(String name)
```

Per ottenere invece tutti i nomi o tutti i valori dei parametri si possono utilizzare i metodi

```
public Enumeration getParameterNames()

public String[] getParameterValues(String name)
```

Infine nel caso di una `GET` HTTP il metodo `getQueryString()` restituisce una stringa con tutti i parametri concatenati.

Nel caso in cui si attendano dati non strutturati e in forma testuale, e l'invocazione sia una `POST`, una `PUT` o una `DELETE` si può utilizzare il metodo `getReader()`, che restituisce un `BufferedReader`.

Se invece i dati inviati sono in formato binario, allora è indicato utilizzare il metodo `getInputStream()`, il quale a sua volta restituisce un oggetto di tipo `ServletInputStream`.

Ecco di seguito un semplice esempio

```
public class MyServlet extends HttpServlet {
   public void service (HttpServletRequest req,
                        HttpServletResponse res) throws ServletException {
      res.setContentType("text/html");
      res.setHeader("Pragma", "no-cache");
      res.writeHeaders();
      String param = req.getParameter("parametro");
      // esegue programma di servizio
      OutputStream out = res.getOutputStream();
      // usa out per scrivere sulla pagina di risposta
   }
}
```

Da notare l'utilizzo dello statement

```
OutputStream out = res.getOutputStream();
```

impiegato per ricavare lo stream di output con il quale inviare dati (ad esempio codice HTML) al client.

I metodi

```
HttpServletRequest.getRealPath(String path)
ServletContext.getRealPath(String path)
```

possono essere utilizzati per determinare la collocazione sul file system per un dato URL: con la versione 2.1 della API il primo di questi due metodi è stato deprecato, consentendo l'utilizzo solo della seconda versione. Il motivo di questa scelta è di praticità da un lato (ridurre la ridondanza), e di correttezza formale dall'altro: la collocazione di una risorsa sul file system dipende infatti dal contesto del Servlet, non da una richiesta.

Per quanto riguarda il formalismo da utilizzare per abbreviare un Uniform Resource Locator, con la versione 2.1 delle API è stata fatta chiarezza: se infatti nella versione precedente erano presenti metodi come `getRequestURL()` ma anche `encodeUrl()`, adesso la regola impone sempre l'utilizzo della lettera maiuscola. Ogni metodo contenente la versione in lettere minuscole è stato deprecato e rimpiazzato.

I metodi interessati sono

```
public boolean HttpServletRequest.isRequestedSessionIdFromUrl()
```

```
public String HttpServletRequest.encodeUrl(String url)
public String HttpServletRequest.encodeRedirectUrl(String url)
```

che diventano così

```
public boolean HttpServletRequest.isRequestedSessionIdFromURL()
public String HttpServletRequest.encodeURL(String url)
public String HttpServletRequest.encodeRedirectURL(String url)
```

Gli headers associati alla chiamata

La codifica dei pacchetti HTTP prevede una parte di intestazione detta header dove si possono codificare le informazioni legate alla tipologia di trasmissione in atto.

Un Servlet può accedere all'header della request HTTP per mezzo dei seguenti metodi della interfaccia `HttpServletRequest`:

```
public String getHeader(String name)
public Enumeration getHeaderNames()
```

Il `getHeader()` permette di accedere all'header dato il nome. Con la versione 2.2 della API è stato aggiunto anche il metodo

```
public Enumeration getHeaders(String name)
```

che permette di ricavare headers multipli i quali possono essere presenti come nel caso nel cache control.

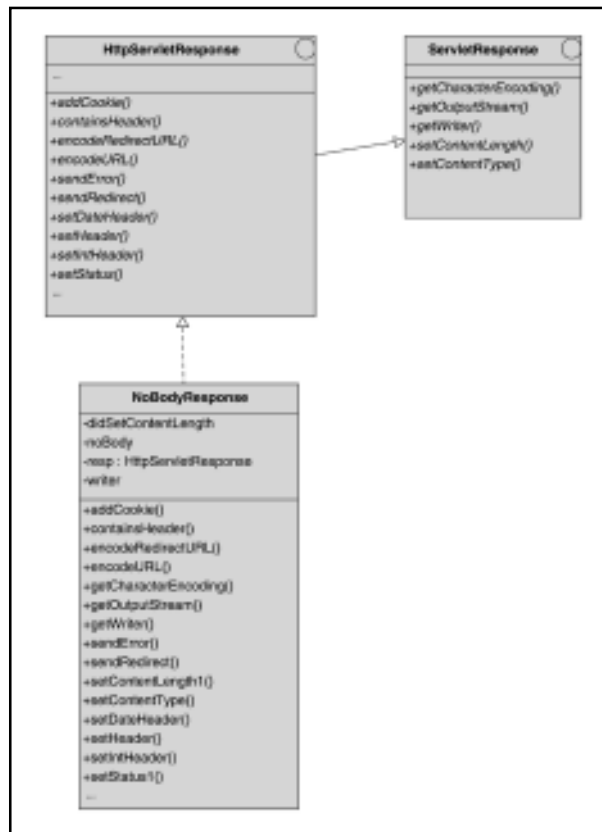
In alcuni casi gli header possono contenere informazioni di tipo non stringa, ovvero numerico o data: in questo caso può essere comodo utilizzare direttamente i metodi

```
public int getIntHeader(String name)
public long getDateHeader(String name)
```

che restituiscono direttamente il tipo più opportuno; nel caso che la conversione a tipo numerico non sia possibile, verrà generata una `NumberFormatException`, mentre se la conversione a data non è possibile, allora l'eccezione generata sarà una `IllegalArgumentException`.

Rispondere è cortesia: `HttpServletResponse`

La risposta del Servlet può essere inviata al client per mezzo di un oggetto di tipo

Figura 14.4 – *Le classi utilizzabili per la gestione delle risposte verso il client*

`HttpServletResponse`, il quale offre due metodi per inviare dati al client: il `getWriter()` che restituisce un oggetto di tipo `java.io.Writer`, ed il `getOutputStream()` che, come lascia intuire il nome, restituisce un oggetto di tipo `ServletOutputStream`.

Si deve utilizzare il primo tipo per inviare dati di tipo testuale al client, mentre un `Writer` può essere utile per inviare anche dati in forma binaria.

Da notare che la chiusura di un `Writer` o di un `ServletOutputStream` dopo l'invio dei dati permette al server di conoscere quando la risposta è completa.

Con i metodi `setContentType()` e `setHeader()` si può stabilire il tipo MIME della pagina di risposta e, nel caso sia HTML, avere un controllo fine su alcuni parametri.

Nell'esempio visto poco sopra tale content-type è stato impostato a `"text/html"`, ad indicare l'invio di una pagina HTML al browser, mentre `"pragma no-cache"` serve per

forzare il browser a non memorizzare la pagina dalla cache (opzione questa molto importante nel caso di pagine a contenuto dinamico).

Gli aspetti legati alla modalità di interazione fra client e Servlet, binaria o testuale, sono molto importanti al fine di comprendere il ruolo della Servlet API in relazione con le altre tecnologie offerte da Java, come ad esempio JSP.

Prima del rilascio della API 2.2, molti server implementavano tecniche di buffering per migliorare le prestazioni (utilizzando memorie tampone tipicamente di circa 8 kilobyte), mentre adesso il cosiddetto *response buffering* è parte integrante della specifica ufficiale.

Cinque nuovi metodi sono stati aggiunti alla interfaccia `ServletResponse` per la gestione diretta del buffer, sia che si utilizzi una comunicazione a mezzo di un `ServletOutputStream` sia di un `Writer`.

Il `getBufferSize()` ad esempio restituisce la dimensione del buffer attualmente utilizzata: nel caso in cui tale risultato sia un intero pari a zero, allora questo significa la mancanza di buffering.

Per mezzo di `setBufferSize ()` il Servlet è in grado di impostare la dimensione del buffer associata ad una risposta: in tal caso è sempre bene specificare una dimensione maggiore di quella utilizzata normalmente dal Servlet, al fine di riutilizzare la memoria già allocata e velocizzare ulteriormente la modifica. Inoltre deve essere invocato prima di ogni operazione di scrittura per mezzo di un `ServletOutputStream` o di un `Writer`.

Il metodo `isCommitted()` restituisce un valore booleano ad indicare se siano stati inviati o meno dei dati verso il cliente. Il `flushBuffer()` invece forza l'invio dei dati presenti nel buffer.

Infine una chiamata delle `reset()` provoca la cancellazione di tutti i dati presenti nel buffer.

Se il buffer è stato appena vuotato a causa dell'invio della risposta verso il client, allora l'invocazione del metodo `reset()` provocherà la generazione di una eccezione di tipo `IllegalStateException`.

Una volta che il buffer viene riempito il server dovrebbe immediatamente effettuare una flush del contenuto verso il client. Se si tratta della prima spedizione dati, si considera questo momento come l'inizio della `commit`.

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse res) throws  ServletException,
                                                   IOException {
   // si imposta un buffer di 8 kb
   res.setBufferSize(8 * 1024);
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();

   // restituisce la dimensione del buffer
   // in questo caso 8192
   int size = res.getBufferSize();
```

```
    // si invia un messaggio nel buffer
    out.println("Messaggio inutile");
    // si ripulisce il buffer eliminando il
    // messaggio precedentemente scritto
    res.reset();

    // si scrive un altro messaggio nel buffer
    out.println("Altro messaggio inutile");

    // si ripulisce il buffer eliminando il
    // messaggio precedentemente scritto
    res.reset();

    // questo messaggio non appare se si invoca la sendError
    out.println("Messaggio probabilmente utile");

    // si esegue un controllo su un parametro importante
    if (req.getParameter("DBName") == null) {
        String MsgError
        = "Manca un parametro indispensabile: il nome del database ";
        res.sendError(res.SC_BAD_REQUEST, MsgError);
    }
}
```

La gestione degli header nella risposta

Nell'inviare la risposta al client, il Servlet può anche modificare l'header HTTP per mezzo dei metodi messi a disposizione dalla interfaccia `HttpServletResponse`: ad esempio il metodo

```
public void setHeader(String name, String value)
```

permette di settare l'header con nome e valore. Nel caso che uno o più valori fossero già presenti con tale nome, allora verrebbero tutti sostituiti con il nuovo valore.

Per aggiungere invece un valore a un nome specifico si può utilizzare il metodo

```
public void addHeader(String name, String value)
```

In alcuni casi i dati memorizzati nell'header HTTP possono essere gestiti meglio direttamente come dati numerici, o tipo data. In questo caso i metodi

```
public void addDateHeader(String name, long date)
public void setDateHeader(String name,long date)
public void addIntHeader(String name, int value)
public void setIntHeader(String name, int value)
```

permettono di effettuare in modo specifico le stesse operazioni di cui prima.

Al fine di inviare in modo corretto le informazioni memorizzate all'interno di un header, ogni operazione di modifica deve essere effettuata prima della commit della risposta, ovvero prima di inviare anche un solo byte verso il client.

Il metodo

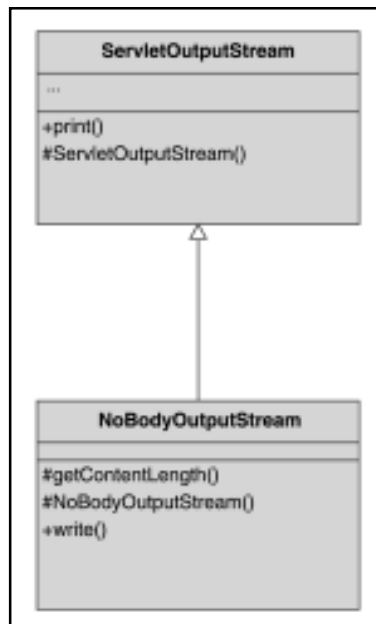
```
public void sendRedirect(String location) throws IOException
```

imposta l'header e il body appropriati in modo da redirigere il client verso un URL differente.

A tal punto eventuali dati memorizzati nel buffer, vengono immediatamente cancellati, dato che la trasmissione viene considerata completa.

Il parametro passato può essere un URL relativo, anche se in tal caso il server dovrebbe sempre effettuare una trasformazione a URL assoluto. Nel caso in cui tale conversione non possa essere fatta per un qualsiasi motivo, allora verrà generata una eccezione di tipo `IllegalArgumentException`.

Anche l'invocazione del metodo `sendError()`, forza il completamento della trasmissione: in questo caso tale metodo permette di specificare un parametro che verrà utilizzato come messaggio di errore nell'header stesso.

Figura 14.5 – *Stream associabili a una risposta*

Analogamente si tenga presente che si ha la medesima situazione nel caso in cui sia stato inviato un quantitativo di informazioni pari a quanto specificato tramite `setContentLength()`.

Con la versione 2.1 della API il metodo

```
HttpServletResponse.setStatus(int sc, String sm)
```

è stato deprecato a favore dei metodi

```
HttpServletResponse.setStatus(int sc)
HttpServletResponse.sendError(String msg)
```

al fine di offrire maggiore eleganza e coerenza nella funzionalità dei metodi.

Distruzione di un Servlet

A completamento del ciclo di vita, si trova la fase di distruzione del Servlet, legata al metodo `destroy()`, il quale permette inoltre la terminazione del processo ed il log dello status.

La ridefinizione di tale metodo, derivato dalla interfaccia `Servlet`, permette di specificare tutte le operazioni simmetriche alla inizializzazione, oltre a sincronizzare e rendere persistente lo stato della memoria.

Il codice che segue mostra quale sia il modo corretto di operare

```
public class myServlet extends HttpServlet {
   public void init(ServletConfig config) throws ServletException {
      // effettua operazioni di inizializzazione
      // per esempio l'apertura della connessione verso il db
   }

   public void destroy() {
      // effettua la chiusura della connessione
   }
}
```

È importante notare che la chiamata della `destroy()`, così come quella della `init()` è a carico del server che ha in gestione il Servlet.

La distruzione di un Servlet e il relativo scaricamento dalla memoria avviene solamente nel momento in cui tutte le chiamate alla `service()` da parte dei client sono state eseguite.

Nel caso in cui tale metodo effettui una computazione più lunga del previsto, il server può dare atto alla distruzione del Servlet, anche se il metodo non ha terminato. In questo caso è compito del programmatore adottare opportune tecniche per lo sviluppo del Servlet, cosa che sarà vista in seguito.

Anche se non strettamente legato al ciclo di vita di un Servlet, occorre conoscere il metodo `getServletInfo()` che di solito viene annoverato fra i metodi principali, accanto a `init()` e `service()`.

Il suo scopo è quello di fornire una descrizione del funzionamento del Servlet: ad esempio il Java Web Server permette di far apparire un messaggio nella finestra di gestione dei vari Servlet installati.

Nel caso in cui si desideri personalizzare il Servlet con un messaggio descrittivo, si può ridefinire il metodo `getServletInfo()`, come ad esempio nella parte di codice che segue

```
public String getServletInfo() {
   return "MyServlet – Un servlet personalizzato – V 1.0"
}
```

L'habitat di un Servlet: il Servlet Context

Il Servlet context rappresenta l'ambiente di esecuzione all'interno del quale il Servlet viene fatto eseguire. L'interfaccia `ServletContext` mette a disposizione una serie di metodi piuttosto utili per interagire con il contesto della applicazione, o per accedere alle risorse messe a disposizione del Servlet stesso.

Ad esempio un Servlet, utilizzando un oggetto di questo tipo può effettuare il log di tutti gli eventi generati, ottenere l'URL associato a tutte le risorse messe a disposizione, e modificare gli attributi ai quali il Servlet può accedere.

Il server su cui il Servlet è in esecuzione è responsabile di fornire una implementazione della interfaccia `ServletContext`, e permette l'utilizzo di una serie di parametri di inizializzazione, che verranno utilizzati dal Servlet all'interno del metodo `init()`.

Con la versione 2.1 della API sono state effettuate una serie di modifiche atte a limitare l'interazione con l'implementazione o il funzionamento del server sottostante: lo scopo è di garantire un più alto livello di sicurezza e correttezza di funzionamento.

Sicuramente una delle modifiche più importanti in tal senso riguarda il metodo

```
SerlvetContext.getServlet(Sring ServletName)
```

il quale permette di ottenere un riferimento a un Servlet correntemente installato nel server e quindi invocarne i metodi pubblici.

Per fare questo è necessario conoscere il nome del Servlet, ottenere accesso all'oggetto `Servlet` del Servlet da invocare, e infine invocare i metodi pubblici del Servlet invocato.

Ad esempio si potrebbe pensare di scrivere

```
public class myServlet1 extends HttpServlet {
```

```
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response) throws ServletException {
     myServlet2 serv2;
     ServletConfig Config = this.getServletConfig();
     ServletContext Context = Config.getServletContext();
     serv2 = (myServlet2) Context.getServlet("servlet2");
     serv2.dosomething();
     serv2.doGet(request, response);
  }
}
```

dove `myServlet2` è la classe del Servlet che si vuole invocare, e `servlet2` è invece il nome logico con cui tale Servlet è stato installato nel server.

Nella la nuova API il metodo `getServlet()` restituisce null poiché il suo utilizzo potrebbe dar vita a scenari inconsistenti nel caso in cui il server abbia invocato il metodo `destroy()` del Servlet: tale metodo infatti è invocabile in ogni momento senza la possibilità o necessità di generare messaggi di avvertimento.

Inoltre, nel caso in cui il server utilizzi tecniche di load balancing grazie all'utilizzo di JVM distribuite, non è possibile sapere in un determinato istante dove il Servlet sia fisicamente in esecuzione.

Sun ha da sempre dichiarato come pericoloso l'utilizzo di tale metodo: la sua presenza si era resa necessaria per permettere la condivisione di informazioni e lo scambio di messaggi fra Servlet differenti, cosa questa che prende il nome di interServlet communication: tale obiettivo potrebbe essere messo in pratica utilizzando ad esempio il pattern Singleton oppure facendo uso di file condivisi.

Una soluzione più elegante e coerente con il resto dell'architettura è offerta dalla API 2.1 che permette di utilizzare il `ServletContext` come repository di oggetti comuni, in modo da condividere informazioni fra i Servlet che sono in esecuzione nello stesso contesto.

Proprio per permettere la gestione dei cosiddetti attributi di contesto, sono stati aggiunti alla `ServletContext` una serie di metodi, fra cui

```
public void setAttribute(String name, Object o)

public Object getAttribute(String name)
```

che permettono di impostare e ricavare un attributo dato il suo nome.

Il secondo era presente fino dalla versione 1.0 delle API, anche se permetteva solamente di ricavare nel server attributi a sola lettura inseriti dal costruttore.

La gestione degli attributi di contesto è molto importante, dato che permette tra le altre cose la comunicazione del Servlet con altri componenti in esecuzione nel container, come ad esempio le pagine JSP.

Completano la lista i metodi

```
public Enumeration getAttributeNames()

public void removeAttribute(String name)
```

i quali consentono rispettivamente di ottenere la lista di tutti i riferimenti, o di rimuoverne uno dato il nome.

Ogni attributo può avere un solo valore; gli attributi che iniziano per `java.` e `javax.` sono riservati, così come quelli che iniziano per `sun.` e `com.sun.`

Sun, per la definizione degli attributi, suggerisce di seguire la regola del dominio rovesciato, adottata anche per la denominazione dei packages.

La comunicazione fra Servlet per mezzo del contesto comune è di grosso aiuto nel caso si debba effettuare dinamicamente la distribuzione dei Servlet fra più JVM, offrendo ai vari Servlet un sistema di comunicazione nativo funzionante anche nel caso in cui la struttura sottostante (le varie JVM) venga modificata.

Come i vari Servlet vengono raggruppati nei vari `ServletContext` dipende dalla configurazione del server. Nel caso dei server più vecchi (che seguono cioè le specifiche precedenti), tutti i Servlet sono collocati nel medesimo `ServletContext`, dato che non vi è differenza fra un contesto ed un altro; adesso invece inizia ad avere senso il raggruppamento e la conseguente suddivisione.

Infine, per poter permettere la comunicazione fra contesti differenti, è stato introdotto il metodo

```
ServletContext.getContext(String urlpath)
```

il quale restituisce un `ServletContext` per un dato URL fornito come parametro.

L'interazione con gli altri oggetti del contesto

Si è già potuto vedere come l'utilizzo degli attributi di contesto permetta l'interazione fra i vari elementi appartenenti allo stesso `ServletContext`.

Con l'avvento della versione 2.1 della Servlet API ed ancor più con la versione 2.2, è stata introdotta la possibilità non solo di scambiare messaggi, ma anche di propagare richieste e risposte fra i vari Servlet dello stesso contesto.

Parallelamente Sun ha introdotto anche altre importanti tecnologie, come Java Server Pages (JSP) ed Enterprise Java Beans (EJB), tecnologie che hanno portato alla nascita del concetto di Web Application: pur rimanendo valido tutto quello che concerne l'interfaccia `ServletContext`, il contesto di un Servlet ora deve essere considerato come un ambiente dove vengono fatti eseguire applicazioni differenti fra loro come Servlet, pagine JSP, componenti EJB. Un Servlet quindi non è più un oggetto isolato, ma collabora con altri elementi in esecuzione nel container.

Una delle innovazioni più interessanti introdotte (con la API 2.1) è la possibilità di delegare la gestione di una richiesta a un altro componente in esecuzione sul server.

Un Servlet può quindi

- inoltrare una richiesta ad un altro componente o Servlet dopo che ne abbia effettuata una prima elaborazione preliminare; questo è uno scenario tipico utilizzato con le JSP;
- includere nella propria risposta quella derivante dalla computazione di un componente esterno.

Queste due operazioni, che rientrano sotto il termine di *request delegation*, permettono di implementare in modo più organico la cosiddetta programmazione server side.

La API 2.1 fornisce una nuova interfaccia, la `RequestDispatcher`, ottenibile per mezzo del metodo

```
ServletContext.getRequestDispatcher(String path)
```

Il reference ottenuto permette di inoltrare la request direttamente all'oggetto identificato dal `path` passato come parametro. La `RequestDispatcher` fornisce i due metodi

```
public void forward(ServletRequest request,
                    ServletResponse response) throws ServletException, IOException

public void include(ServletRequest request,
                    ServletResponse response) throws ServletException, IOException
```

Il primo permette di inoltrare una richiesta pervenuta: al fine di garantire che il controllo passi totalmente al secondo oggetto, è necessario che l'invocazione della `forward()` avvenga prima dell'ottenimento di un `ServletOutputStream` o della creazione di un `Writer` da utilizzare per l'invio dei dati verso il client.

La chiamata alla `include()` invece può essere fatta in ogni momento, dato che il controllo resta al Servlet chiamante.

Ecco una breve porzione di codice in cui viene mostrato come includere l'output di un altro Servlet

```
// mostra un primo output di preparazione
out.println("Ecco la risposta della interrogazione:");

RequestDispatcher dispatcher = getServletContext();
dispatcher.getRequestDispatcher("/servlet/DBServlet?param=value");
dispatcher.include(req, res);
out.println("Si desidera effettuare una nuova ricerca?");
```

In questo caso il primo Servlet si serve di un secondo, il `DBservlet`, per effettuare le ricerche in un database relazionale. Il secondo Servlet emette una risposta automaticamente inviata al client, che riceve successivamente anche la risposta del primo Servlet.

Invece di invocare il secondo Servlet per mezzo della chiamata all'URL passando i parametri di esecuzione, per inoltrare una richiesta si può in alternativa settare manualmente gli attributi della `ServletRequest` per mezzo del metodo `ServletRequest.setAttribute()`.

Ad esempio

```
// mostra un primo output di preparazione
RequestDispatcher dispatcher = getServletContext();
out.println("Ecco le possibili risposte");
dispatcher.getRequestDispatcher("/servlet/DBServlet");

out.println("La prima risposta ");
// effettua una prima invocazione
String value1 = ...
req.setAttribute("param", value1);
dispatcher.include(req, res);

out.println("La seconda risposta ");
// effettua una seconda invocazione
String value2 = ...
req.setAttribute("param", value2);
dispatcher.include(req, res);

out.println("Si desidera effettuare una nuova ricerca?");
```

In questo caso il Servlet chiamato può ricavare il valore dell'attributo legato alla invocazione per mezzo di una istruzione del tipo

```
String value1 = req.getAttribute("param");
```

o , in maniera equivalente, per mezzo di una più generica

```
String[] values = req.getAttributeNames();
```

Il vantaggio di utilizzare gli attributi invece delle semplici query string permette di passare come parametri oggetti al posto di semplici stringhe.

Con la API 2.2 è stato anche aggiunto alla interfaccia `ServletContext` il metodo

```
public RequestDispatcher getNamedDispatcher(java.lang.String name)
```

che permette di inoltrare una richiesta ad un componente specificato per mezzo del suo

nome univoco, piuttosto che utilizzando un URL. È questa una funzionalità resa possibile grazie al concetto di container e permette di inoltrare le richieste verso oggetti che non sono resi pubblici tramite un indirizzo.

Nella interfaccia `ServletRequest`, il metodo

```
public RequestDispatcher getRequestDispatcher(String path)
```

è stato modificato in maniera da accettare un URL relativo come parametro, differentemente dal `getRequestDispatcher()` il quale invece accetta solamente URL completi.

Nel caso di Web Applications l'URL relativo può essere utilizzato, ad esempio, per inoltrare richieste a oggetti facenti parte dello stesso contesto.

Anche il metodo

```
public void sendRedirect(String location)
```

è stato modificato in modo da accettare URL relativi, che verranno anche in questo caso modificati direttamente dal Servlet container aggiungendo l'indirizzo URL del contesto dove viene eseguito il Servlet.

Il context e le applicazioni distribuite

Sempre con la API 2.2 è stato ridefinito e standardizzato il funzionamento del server in modo da permettere il passaggio di oggetti condivisi fra VM distribuite, senza la generazione di indesiderate `ClassCastException.` Questo inconveniente poteva generarsi quando Servlet differenti venivano caricati da classloaders diversi, dato che due oggetti caricati in memoria da due loader diversi non possono essere "castati" l'un l'altro.

In base a tale regola, la modifica effettuata su un Servlet, oltre che implicare il ricaricamento in memoria del Servlet stesso, provocherà anche il reload di tutte le classi da esso utilizzate: questo da un lato può comportare una diminuzione delle prestazioni, ma ha il grosso pregio di permettere la sincronizzazione del codice e di semplificare non poco la fase di sviluppo delle applicazioni.

Infine si tenga presente che la specifica 2.2 ribadisce che nel caso di applicazioni distribuite un contesto deve essere utilizzato per una sola JVM: i Context Attributes non possono essere quindi utilizzati per memorizzare attributi a scope globale. Questo obiettivo può essere perseguito utilizzando risorse esterne al server, come ad esempio database o componenti EJB.

Una applicazione marcata come distribuibile segue inoltre altre importanti regole per le sessioni utente: i server infatti utilizzano la cosiddetta *session affinity* per una gestione efficace del mantenimento dello stato fra i vari server. Questo implica che tutte le richieste a una singola sessione da parte di un certo utente sono gestite da una singola JVM alla

volta, cosa che tra l'altro evita di dover sincronizzare fra i vari server le informazioni memorizzate da una certa sessione.

Visto che il meccanismo utilizzato per trasferire i dati è al solito quello della serializzazione, le variabili memorizzate nelle varie sessioni devono essere serializzabili,. Nel caso in cui un oggetto non sia serializzabile il server produrrà una `IllegalArgumentException`.

Resource Abstraction

Con la versione 2.1 della API è stato formalizzato il concetto di astrazione delle risorse, consentendo così l'accesso alle risorse di sistema indipendentemente dalla loro collocazione nel sistema, essendo i vari riferimenti gestiti sotto forma di URL.

Questo permette di gestire i Servlet come oggetti indipendenti dal contesto e che possono essere spostati da un server a un altro senza particolari difficoltà o senza la necessità di dover riscrivere buona parte del codice.

Il metodo che permette di ottenere una risorsa è il

```
ServletContext.getResource(String uripath)
```

dove `uripath` rappresenta l'URI in cui è collocata tale risorsa: è compito del web server effettuare il mapping fra risorsa vera e propria e URL.

Una risorsa astratta non può essere una risorsa attiva (Servlet, CGI, JSP, programmi o simili), per le quali si deve utilizzare la `RequestDispatcher`. Ad esempio una invocazione `getResource("/index.jsp")` ritorna il sorgente del file JSP e non il risultato del processamento di tale file da parte del server.

Una volta ottenuto il riferimento all'URL della risorsa è possibile effettuare le operazioni permesse su tale risorsa: ad esempio si può pensare di individuare una pagina HTML e di effettuarne successivamente la stampa verso il client.

```
String res="/static_html/header.html";
URL url = getServletContext().getResource(res);
out.println(url.getContent());
```

Qui il file `header.html` rappresenta la prima parte della pagina HTML da inviare al client, e può essere collocato sotto la web root sul server dove viene eseguito il Servlet, o in alternativa su un server remoto.

Una risorsa astratta possiede una serie di caratteristiche che possono essere conosciute utilizzando una connessione verso l'URL relativo alla risorsa; ad esempio

```
// ricava l'url della front page
URL url = getServletContext().getResource("/");
URLConnection con = url.openConnection();
```

```
con.connect();
int ContentLength = con.getContentLength();
String ContentType = con.getContentType();
long Expiration = con.getExpiration();
long LastModified = con.getLastModified();
```

Il metodo

```
InputStream in = getServletContext().getResourceAsStream("/")
```

permette di ottenere direttamente uno stream associato alla risorsa remota, senza doverlo aprire esplicitamente come ad esempio

```
URL url = getServletContext().getResource("/");
InputStream in = url.openStream();
```

Nel caso in cui la risorsa lo permetta, è possibile utilizzare il metodo `getResource()` per ottenere un riferimento in scrittura su una risorsa; ad esempio

```
URL url = getServletContext().getResource("/myservlet.log");
URLConnection con = url.openConnection();
con.setDoOutput(true);
OutputStream out = con.getOutputStream();
PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
pw.println("Evento verificatosi il" + (new Date());
pw.close();
out.close();
```

Nel caso non si possa o non si voglia utilizzare esclusivamente la API 2.1, si potrà continuare a effettuare il riferimento alle risorse di tipo file utilizzando i metodi standard per l'accesso a file, come ad esempio `getPathTranslated()`, anche se in questo caso un oggetto di tipo `File` è strettamente legato alla macchina e non è portabile.

Con la versione 2.2 delle API è stato aggiunto al `ServletContext` l'attributo

```
javax.servlet.context.tempdir
```

che contiene il nome della directory temporanea utilizzata dal server: tale informazione può essere utilizzata, tramite un oggetto di tipo `java.io.File`, per effettuare operazioni di I/O nella directory temporanea di cui sopra. Ecco un esempio

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse res) throws ServletException,IOException{
```

```
    // ricava la directory temporanea come un object File
    String Attr="javax.servlet.context.tempdir";
    Object o = getServletContext().getAttribute(Attr);
    File dir = (File) o;

    // Crea un file temporaneo nella temp dir
    File f = File.createTempFile("xxx", ".tmp", dir);

    // inzia le operazioni di scrittura
    FileOutputStream fos = new FileOutputStream(f);

}
```

Sia durante la fase di debug di un Servlet, così come in esecuzione, può essere molto utile controllare il funzionamento del Servlet ed eventualmente effettuare il log delle operazioni effettuate.

È per questo motivo che il `ServletContext`, per effettuare il salvataggio di messaggi di log, mette a disposizione il metodo

```
log(Exception e, String msg)
```

(disponibile fino alla versione 2.1 della Servlet API), con il quale è possibile memorizzare un messaggio relativo a una generica `Exception`.

Con il passaggio alla versione 2.1 della API, tale metodo è stato deprecato, e sostituito con

```
log(String msg, Throwable e)
```

Questo cambiamento, oltre ad essere maggiormente in linea con lo standard Java che vuole il posizionamento delle eccezioni in ultima posizione fra i vari parametri, consente l'utilizzo di una eccezione più generica della semplice `Exception`.

Inoltre il metodo è stato spostato nella classe `GenericServlet`, cosa che permette una sua invocazione del log senza dover prima ricavare un riferimento al `ServletContext`.

Il context e la versione della API utilizzata

Dato che l'evoluzione della Servlet API, pur dando vita di versione in versione ad un framework sempre più robusto, potente e coerente, introduce non pochi problemi di compatibilità. Anche se per fortuna le variazioni sono in genere fatte in modo intelligente in modo da garantire la retrocompatibilità, può essere utile in certi casi ricavare la versione della API utilizzata: dalla 2.1 sono disponibili i seguenti metodi che restituiscono questo genere di informazioni

```
public int getMajorVersion()

public int getMinorVersion()
```

i quali, nel caso della versione 2.1 restituiscono rispettivamente i due interi `2` e `1`.

I Servlet e l'internazionalizzazione

La possibilità di rendere una applicazione sensibile alla localizzazione geografica in cui viene eseguita è sicuramente una delle caratteristiche più interessanti che Java mette a disposizione grazie alla internazionalizzazione.

Nel caso della programmazione Web questa caratteristica è sicuramente ancor più importante, dato che un client potrebbe essere ubicato in una qualsiasi regione del mondo connessa alla rete.

È per questo motivo che, oltre al normale supporto per la localizzazione nel caso della Servlet API 2.2 sono stati aggiunti alcuni metodi alla interfaccia `HttpServletRequest` con lo scopo di determinare la localizzazione del client e quindi agire di conseguenza.

Il metodo `getLocale()` restituisce un oggetto di tipo `java.util.Locale` in base alle informazioni memorizzate nell'header `Accept-Language` della richiesta.

Il metodo `getLocales()` invece restituisce la lista di tutti i `Locale` accettati dal client con il preferito in testa alla lista.

Parallelamente il metodo `setLocale()` della `HttpServletResponse` permette di impostare il `Locale` opportuno per inviare la risposta nel modo migliore verso il client.

Ad esempio questi metodi potrebbero essere utilizzati nel seguente modo

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse res) throws ServletException,IOException {

   res.setContentType("text/html");
   Locale locale = req.getLocale();
   res.setLocale(locale);
   PrintWriter out = res.getWriter();

   // scrive un output sulla base della lingua
   // ricavata tramite locale.getLanguage()
}
```

Invocazione di Servlet

Come si è potuto comprendere nella sezione dedicata al ciclo di vita di un Servlet, esso "vive" solamente durante l'invocazione di una `GET` o di una `POST` HTTP: è quindi importante capire come sia possibile effettuare una invocazione con parametri da parte del client.

Di seguito saranno quindi prese in esame le principali modalità di invocazione, che devono essere aggiunte al caso di invocazione di Servlet da Servlet.

Invocazione diretta dell'URL

La differenza fondamentale fra una GET ed una POST sta nella modalità con cui viene effettuata l'invocazione e soprattutto nel modo con cui sono passati i parametri. Nel primo caso il tutto avviene in modo diretto tramite la composizione dell'URL di invocazione della risorsa remota (Servlet o altro) appendendo i parametri dopo il carattere ?. In questo caso la lunghezza massima di una invocazione di questo tipo deve essere di 256 caratteri. In una POST invece i parametri sono passati come elementi dell'header del pacchetto HTTP.
In questo caso non si ha nessuna limitazione sulla lunghezza della invocazione, e tutta la comunicazione avviene in modo invisibile, differentemente dalla GET dove i parametri appaiono in chiaro nella URL di invocazione.
Vi è però un'altra importante differenza: nelle GET la chiamata rimane in memoria nella cache del browser o può essere salvata nella memoria del web server al fine di ridurre il tempo di latenza da parte del client.
Una POST invece non viene mai salvata in alcun tipo di memoria: è questa una precisa scelta progettuale e può essere sfruttata per aumentare il livello di sicurezza nel caso in cui, ad esempio, serva una estrema riservatezza sui dati.

Il modo più semplice per invocare un Servlet è quello di invocare l'URL corrispondente al Servlet, ad esempio tramite browser. Questo tipo di invocazione corrisponde ad una GET HTTP ed ha la seguente sintassi

```
http://nomehost:porta/nomeservlet[?parametro1=valore1&parametro2=valore2]
```

ad esempio

```
http://www.mysite.com/myservice?url=myurl&user=franco
```

In questo caso i parametri sono passati direttamente al Servlet per mezzo della sintassi particolare dell'invocazione del Servlet.

Il server potrebbe aver abilitato un alias con nome simbolico, per cui l'invocazione potrebbe prendere la forma

```
http://www.mysite.com/myservice.html?url=myurl&user=franco
```

Indipendentemente dalla modalità di invocazione, il codice necessario per ricavare un parametro passato dall'interno del codice Java, è sempre lo stesso, ovvero per mezzo del metodo `getParameter()`.

Un'invocazione di tipo `GET` corrisponde ad una comunicazione di messaggi di testo della lunghezza massima di 256 caratteri, che diventa la lunghezza massima della stringa da passare come parametro al Servlet.

Invocazione per mezzo di un form HTML

Tramite la realizzazione di un form HTML è possibile invocare il Servlet specificandone l'URL associato. In questo caso oltre ai parametri da passare al Servlet si può indicare la tipologia di invocazione (`GET` o `POST`).

Ad esempio un form HTML potrebbe essere

```
<FORM METHOD="POST" ACTION=="http://www.mokabyte.it/servlet/MyServlet">
        Immetti un testo: <BR> <INPUT TYPE="text" NAME="data"><BR>
     <input type="submit" name="B1" value="ok">
  </FORM>
```

che dà luogo a un form con una casella di testo (il cui contenuto poi verrà inviato al Servlet sotto forma di parametro denominato in questo caso *data*), e di un pulsante con etichetta "OK" (fig. 14.6).

In questo caso il Servlet viene invocato effettivamente alla pressione del pulsante del form: il browser provvede a raccogliere tutti i dati ed inviarli al Servlet il cui URL è specificato per mezzo del tag `ACTION`.

Figura 14.6 – *Un tipico form HTML per l'invocazione di un Servlet*

L'esempio che segue mostra come sia possibile ricavare tutti i parametri passati al Servlet per mezzo di una invocazione tramite form.

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;


public class SimpleFormServlet extends HttpServlet {

public void service(HttpServletRequest req,
                    HttpServletResponse res) throws IOException {

      Enumeration keys;
      String key;
      String value;
      String title;

      res.setContentType("text/html");
      ServletOutputStream out = res.getOutputStream();
      out.println("<HEAD><TITLE>");
      out.println("SimpleFormServletOutput");
      out.println("</TITLE></HEAD><BODY>");

      out.println("<h1> Questo è l'output del servlet");
      out.println("SimpleFormServlet </h1>");

      // ricavo i nomi e i valori dei parametri
      keys = req.getParameterNames();
      while (keys.hasMoreElements()) {
         key = (String) keys.nextElement();
         value = req.getParameter(key);
         out.println("<P><B>");
         out.print("Nome Parametro: </B>");
         out.print(key);
         out.print("<BR> <B> valore:</B>");
         out.print(value);
      }
      out.println("</BODY>");
   }

   public String getServletInfo() {
      String msg = "SimpleFormServlet"
      msg = msg + "ricava la lista di tutti i parametri inviati ";
      msg = msg + "tramite form ed invia la lista client ";
      return msg;
   }
```

```
}
```

In questo caso i parametri sono ricavati tutti, e tramite pagine HTML viene inviata una risposta al client contenente tutti i nomi ed i valori passati.

Il codice HTML della pagina che invoca tale Servlet è piuttosto banale: ad esempio si potrebbe avere

```
<form METHOD=POST ACTION="http://www.mokabyte.it/form">
<input TYPE=hidden SIZE=30 NAME="nascosto" VALUE="messaggio segreto">
<table>
<tr>
<td>Nome</td>
<td><input TYPE=text SIZE=30 NAME="nome"></td>
</tr>

<tr>
<td>Cognome</td>
<td><input TYPE=text SIZE=30 NAME="cognome"></td>
</tr>

<tr>
<td>Indirizzo</td>
<td><input TYPE=text SIZE=30 NAME="indirizzo"></td>
</tr>

...

</table>

<p><input TYPE="submit" VALUE=" ok "><input TYPE="reset" VALUE="Clear Form"></center>
</form>
```

Si noti la presenza di un parametro nascosto, così come dei due tag input con `TYPE` uguale a `"submit"` (è il pulsante che attiva l'invocazione), e di `"reset"` (il pulsante che pulisce il form). Infine si noti che l'action associata a tale form corrisponde ad un serlvet *mappato* con nome logico `form`.

Il tag `SERVLET`

Un modo diverso di utilizzare i Servlet per creare pagine HTML in modo dinamico è quello basato sul tag `SERVLET`, tag che deve essere inserito all'interno del codice HTML. In questo caso la pagina (che si chiama server side e che deve avere estensione `.shtml`), viene preprocessata prima di essere inviata al client: in tale fase il codice HTML contenuto all'interno della coppia `<SERVLET> ... </SERVLET>` viene sostituito con l'output prodotto dal Servlet.

Il web server deve poter supportare il tag SERVLET (possibilità in genere specificata nelle caratteristiche del prodotto), altrimenti la pagina non verrà modificata, inviando al client il codice HTML originario.

Con l'introduzione delle tecnologia JSP, che può esserne considerata l'evoluzione, tale tecnica non viene più utilizzata, sia per la limitazione e minor potenza espressiva rispetto a JSP, sia e soprattutto perché è poco efficiente: per ogni invocazione da parte del client infatti il server deve processare la pagina HTML ed inserire il risultato del Servlet, senza la possibilità di eseguire alcuna operazione di compilazione o memorizzazione come avviene invece in JSP. Un esempio di pagina HTML server side potrebbe essere

```
<HTML>
<BODY>

<!— parte grafica HTML... -->
<p>The response is: <i>
   <SERVLET NAME=MyServlet>
   <PARAM NAME=param1 VALUE=val1>
   <PARAM NAME=param2 VALUE=val2>
   </SERVLET>
</i></p>
<!— parte grafica HTML... -->
</BODY>
</HTML>
```

Per quanto riguarda l'implementazione del Servlet, non ci sono particolari differenze rispetto agli altri casi; se non per il fatto tassativo che il Servlet invii una risposta al client.

Terminazione corretta dei Servlet

Può accadere che i metodi di servizio, ovvero quelli che rispondono alle invocazioni del client, primo fra tutti il service(), non abbiano terminato la loro attività al momento in cui il server invoca la destroy().

Tecnicamente questa situazione, che prende il nome di *potentially long-running service requests*, deve essere opportunamente gestita tramite alcune operazioni, fra cui:

- tenere traccia del numero di thread correntemente in esecuzione durante una chiamata al metodo service();
- implementare il cosiddetto shutdown pulito, permettendo al metodo destroy() del Servlet di notificare i vari thread dello shutdown in atto, e permettere loro la fine del lavoro;

- eventualmente effettuare check nei vari punti a maggiore carico computazionale, in modo da simulare uno shutdown, e quindi verificare la costante responsività dei thread a lunga computazione.

Per tenere traccia delle richieste da parte dei client, si può ad esempio utilizzare una variabile nel Servlet che svolga proprio la funzione di contatore. Questa variabile deve essere accessibile sia per la lettura che per la scrittura, attraverso metodi sincronizzati. Ad esempio si potrebbe pensare di scrivere

```
public myServletShutdown extends HttpServlet {
   // variabile privata che memorizza il numero di thread attivi
   private int serviceCounter = 0;

   // metodo per l'incremento del numero di thread attivi
   // è sincronizzato per garantire la coerenza
   protected synchronized void addRequest () {
      serviceCounter++;
   }

   // metodo per il decremento del numero di thread attivi
   // è sincronizzato per garantire la coerenza
   protected synchronized void removeRequest () {
      serviceCounter--;
   }

   // metodo per ottenere il numero di thread attivi
   // è sincronizzato per permettere di ricavare un  valore coerente
   protected synchronized int getTotRequest() {
      return serviceCounter;
   }
}
```

In questo esempio la variabile `serviceCounter` verrà incrementata ogni volta che viene effettuata una chiamata da parte del client, e decrementata quando il metodo `service()` termina. In questo caso il `service()` può essere modificato nel seguente modo

```
public void service(HttpServletRequest req,
                  HttpServletResponse resp) throws ServletException {
   addRequest();
   try {
      super.service(req, resp);
   }
   finally {
      removeRequest();
   }
}
```

Come si può notare in questo caso la `service()` effettua una invocazione alla `service()` del Servlet padre, effettuando però sia prima che dopo le modifiche alla variabile contatore.

Per effettuare invece lo shutdown indolore del Servlet è necessario che il metodo `destroy()` non distrugga ogni risorsa condivisa indistintamente, senza controllare se questa sia utilizzata da qualcuno.

Un primo controllo, necessario ma non sufficiente, da effettuare è quello sulla variabile `serviceCounter` come nell'esempio visto poco sopra. Infatti si dovrà notificare a tutti i thread in esecuzione di un task particolarmente lungo che è arrivato il momento di concludere il lavoro.

Per ottenere questo risultato, si può far ricorso a un'altra variabile al solito accessibile e modificabile per mezzo di metodi sincronizzati.

Ad esempio

```
public mySerlvetShutdown extends HttpServlet {
   private boolean shuttingDown;
   ...
   // Metodo di modifica per l'attivazione
   // o disattivazione dello shutdown
   protected setShuttingDown(boolean flag) {
      shuttingDown = flag;
   }

   // metodo per la lettura dello stato dello shutdown
   protected boolean isShuttingDown() {
      return shuttingDown;
   }
}
```

Un esempio di utilizzo della procedura pulita per lo shutdown potrebbe essere ad esempio quella mostrata qui di seguito

```
public void destroy() {

   // controlla se vi sono ancora servizi in esecuzione
   // e in tal caso invia un messaggio di terminazione
   if (numServices() > 0) {
      setShuttingDown(true);
   }

   // Attende lo stop dei vari servizi
   while(numServices() > 0) {
      try {
         Thread.sleep(interval);
      }
```

```
      catch (InterruptedException e) {
         System.out.println("Si è verificata una eccezione");
      }
   }
}
```

Questa porzione di codice, che potrebbe apparire per certi versi piuttosto oscura, assume significato se si prende in considerazione quella che è la terza raccomandazione da seguire, ovvero quella di implementare sempre una politica corretta (nel senso del termine inglese *polite*) di gestione dei vari thread.

Secondo tale regola tutti i metodi di thread che possono effettuare computazioni lunghe dovrebbero di tanto in tanto controllare lo stato della variabile `shuttingDown` e, se necessario, interrompere l'esecuzione del task attivo in quel momento.

Ad esempio si può pensare di scrivere

```
public void doPost(...) {
   // esegue una computazione molto lunga
   for(i = 0; ((i < BigNumber) && !isShuttingDown()); i++) {
      try {
      // esegue una qualsiasi operazione costosa
      executionTask(i);
      }
      catch (InterruptedException e) {
         System.out.println("Si è verificata una eccezione ");
      }
   }
}
```

In questo caso l'esecuzione del lavoro a lungo termine viene eseguita direttamente all'interno del metodo `doPost`, piuttosto che in un thread separato. Questa soluzione, che per certi versi potrebbe apparire non troppo elegante, ha il vantaggio, come si è potuto vedere, di permettere un controllo maggiore circa l'esecuzione delle varie parti.

Il mantenimento dello stato

Per mantenimento dello stato si intende l'utilizzo di una qualche tecnica che permetta di mantenere informazioni fra due o più invocazioni successive di un Servlet.

Al termine della esecuzione del metodo `service()` infatti il Servlet perde il riferimento del client specifico e non è in grado di utilizzare informazioni prodotte precedentemente.

Si immagini ad esempio il caso in cui un client debba effettuare due o più invocazioni dello stesso Servlet, in modo tale che dopo la prima invocazione le successive debbano utilizzare i valori prodotti come risposta della prima invocazione.

Si dovrà allora predisporre un sistema che salvi i risultati intermedi o sul client, oppure sul server in un qualche registry, associando però ad ogni entry nel registry una chiave che identifichi il client a cui fa riferimento.

Di seguito sono riportate le principali soluzioni adottate per ottenere questo scopo.

Modifica della invocazione

Si supponga ad esempio il caso in cui il client effettui una prima invocazione del Servlet inviando alcuni parametri per mezzo di un form HTML. Il Servlet invierà quindi in risposta al client una pagina HTML contenente un secondo form con il quale il client potrà effettuare una seconda invocazione. Si vuole che questa seconda invocazione tenga conto anche della prima, ovvero che sia mantenuta memoria dei parametri della prima invocazione. Dato che l'HTTP è un protocollo stateless, un modo per fare questo potrebbe essere di modificare la pagina HTML inviata in risposta alla prima interrogazione.

Si supponga ad esempio di avere un form HTML il cui codice potrebbe essere quello visto in precedenza

```
<FORM METHOD="POST" ACTION=="http://www.mokabyte.it/servlet/myservlet">
   Immettere lo user name<INPUT TYPE="text" NAME="user_id"><BR>
   <input type="submit" name= "articolo" value="ok">
</FORM>
```

Se dopo la prima invocazione il Servlet vorrà memorizzare nel client per le invocazioni successive il valore `"user_id=franco98"`, calcolato alla prima invocazione, allora potrà modificare la pagina di risposta in modo che sia

```
<FORM METHOD="POST" ACTION="http://www.mokabyte.it/servlet/MyServlet">
   Quali articoli vuoi comprare? <INPUT TYPE="text" NAME="data"><BR>
   <input type="submit" name="articolo" value="ok">
   <input type="hidden" name="user_id" value="franco98">
</FORM>
```

In questo caso il tag HTML `hidden` consente la creazione di un campo non visibile nella finestra del browser, ma che andrà a comporre a tutti gli effetti il set dei parametri dell'invocazione successiva.

Discorso del tutto analogo nel caso in cui la pagina di risposta non contenga un form, ma un semplice link per l'invocazione successiva del Servlet: in questo caso infatti sarà necessario modificare il link presente nella pagina; ad esempio da

```
http://www.mokabyte.it/servlet/MyServlet?parametro_nuovo=valore
```

a

```
http://www.mokabyte.it/servlet/
   MyServlet?parametro_vecchio=valore&parametro_nuovo=valore
```

Cookie

La tecnica precedente rappresenta il modo più rudimentale per memorizzare sul client una serie di informazioni fra due invocazioni successive: oltre al fatto che è possibile memorizzare esclusivamente oggetti di tipo stringa, tale tecnica ha la grossa limitazione di mantenere in chiaro i valori memorizzati (sono direttamente visibili controllando il codice HTML), offrendo quindi un livello di sicurezza certamente molto basso.

Una prima alternativa più evoluta è quella basata sull'utilizzo dei cookie, oggetti ben noti alla maggior parte dei navigatori internet a causa anche delll'impopolarità che hanno riscosso nei primi tempi.

Un cookie permette la memorizzazione di informazioni nella memoria del browser: il Servlet può inviare un cookie al browser aggiungendo un campo nell'header HTTP . In modo del tutto simmetrico, leggendo tale campo è in grado di prelevare i cookie memorizzati.

Prendendo ad esempio il caso della implementazione di un carrello virtuale della spesa, caso peraltro piuttosto frequente, si potrebbe avere un cookie `ArticleInBasket` che assume il valore `231_3`, ad indicare che sono stati aggiunti al carrello 3 oggetti il cui codice sia 231.

È plausibile inoltre avere per lo stesso nome di cookie valori multipli: ad esempio accanto al valore `231_3`, il Servlet potrebbe memorizzare anche il valore `233_4`, sempre con l'obiettivo di memorizzare l'aggiunta al carrello di 4 istanze dell'oggetto il cui codice sia 234.

Secondo le specifiche del protocollo HTTP un browser dovrebbe permettere la memorizzazione di almeno 20 cookie per host remoto, di almeno 4 kilobyte ciascuno. È questa una indicazione di massima, ed è comunque consigliabile far riferimento alle specifiche dei vari browser. In alcuni casi il server, a seconda delle impostazioni utente e delle caratteristiche del server stesso, può ottenere solamente i cookie da lui memorizzati nel client: in questo caso tutti i Servlet in esecuzione all'interno dello stesso server possono di fatto condividere i vari cookie memorizzati.

In Java si possono creare e gestire i cookie tramite la classe `javax.servlet.http.Cookie`: ad esempio per la creazione è sufficiente invocare il costruttore

```
Cookie C = new Cookie("uid","dedo12")
```

In tal caso si memorizza un cookie nel browser in modo da tener memoria della coppia variabile, valore `"uid=franco98"`. Eventualmente è possibile modificare successivamente tale valore per mezzo del metodo `setValue()`.

Il nome di un cookie deve essere un token dello standard HTTP/1.1: in questo caso per token si intende una stringa che non contenga nessuno dei caratteri speciali menzionati nella RFC 2068, o quelli che iniziano per `$` che sono invece riservati dalla RFC 2109.

Il valore di un cookie può essere una stringa qualsiasi, anche se il valore null non garantisce lo stesso risultato su tutti i browser. Se si sta inviando un cookie conforme con la specifica originale di Netscape, allora non si possono utilizzare i caratteri

```
[ ] ( ) = , " / ? @ : ;
```

È possibile inoltre settare alcuni parametri e informazioni del cookie stesso, come il periodo massimo di vita, metodo `setMaxAge()`, periodo dopo il quale il cookie viene eliminato dalla cache del browser, oppure inserire un commento tramite il metodo `setComment()` che verrà presentato all'utente nel caso questo abbia abilitato il controllo: questa non è una feauture standard e quindi non è trattata nello stesso modo da tutti i browser.

Una volta creato, per aggiungere il cookie al browser si può utilizzare il metodo

```
HttpServletResponse.addCookie(C)
```

Ad esempio per creare un cookie ed inviarlo al browser

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response) throws ServletException,
                                                       IOException {
   Cookie getItem = new Cookie("Buy", ItemId);
   getBook.setComment("Oggetto acquistato dall'utente ");
   response.addCookie(getBook);
}
```

Nel caso in cui il Servlet restituisca una risposta all'utente per mezzo dell'oggetto `Writer`, si deve creare il cookie prima di accedere al `Writer:` infatti, dato che i cookie sono inviati al client come variabili dell'header le operazioni relative all'header stesso devono essere effettuate, affinché abbiano effetto, prima dell'invio dei dati veri e propri.

Ad esempio si dovrà scrivere

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response) throws ServletException,
                                                       IOException {

   // crea un cookie
   Cookie getItem = new Cookie("Buy", ItemId);

   // imposta il content-type header prima di acccedere al Writer
```

```
   response.setContentType("text/html");

   // ricava il Writer e scrive i dati della risposta
   PrintWriter out = response.getWriter();
   out.println("<html><head><title> Book Catalog </title></head>");
}
```

L'operazione opposta, ovvero ricavare i vari cookies memorizzati nel browser, può essere effettuata grazie ai metodi

```
public Cookie getCookies(String nomecookie)

public Cookie[] getCookies()
```

Il primo restituisce un cookie identificato per nome, mentre il secondo tutti quelli memorizzati ed è presente nella classe `javax.servlet.http.HttpServletRequest`.

```
Cookie[] cookies = request.getCookies();
for(i=0; i < cookies.length; i++) {
   Cookie thisCookie = cookie[i];
   if (thisCookie.getName().equals("compra")) {
      System.out.println("trovato il cookie " + thisCookie.getValue());
      // rimuove il cookie impostando l'età massima a 0
      thisCookie.setMaxAge(0);
   }
}
```

In questo caso dopo aver ricavato tutti i cookie disponibili, si stampa il valore in essi memorizzato per mezzo del metodo `getValue()`; si noti come sia possibile invalidare un cookie semplicemente impostando la sua età massima a zero.

Si deve dire che per quanto flessibile e semplice, la soluzione dei cookie spesso non viene scelta a causa del basso livello di sicurezza offerto. Infatti il cookie e quindi anche le informazioni in esso contenute, sono memorizzate sul client: questa situazione può consentire a malintenzionati di ricavare tali informazioni durante il loro tragitto dal client al server (ad esempio durante una invocazione).

Inoltre se mentre in una intranet è abbastanza plausibile aspettarsi che tutti i browser supportino i cookie o comunque siano configurati per usarli, una volta in internet non si può più fare affidamento su tale assunzione, per cui vi possono essere casi in cui tale tecnica può non funzionare.

Le sessioni

Il concetto di sessione, introdotto da Java proprio con i Servlet permette di memorizzare le informazioni su client in modo piuttosto simile a quanto avviene ad esempio per i

cookie: la differenza fondamentale è che le informazioni importanti sono memorizzate all'interno di una sessione salvata sul server, invece che sul client, al quale viene affidato solamente un identificativo di sessione.

Dato che non si inviano dati al client come invece avviene con i cookie, si ha una drastica riduzione del traffico di dati in rete con benefici sia per le prestazioni che per la sicurezza.

Per associare il client con la sessione salvata, il server deve "marcare" il browser dell'utente: questo viene fatto inviando un piccolo e leggero cookie sul browser con il numero di identificazione della sessione, il Session Id.

Le interfacce che permettono la gestione delle sessioni sono la `HttpSession`, la `HttpSessionBindingListener` e la `HttpSessionContext`.

Per creare una session, si può utilizzare il metodo `getSession()` della classe `HttpServletRequest`: come ad esempio:

```
public class SessionServlet extends HttpServlet {

public void doGet (HttpServletRequest request,
                   HttpServletResponse response)  throws ServletException,
                                                         IOException {
   // ricava o crea la sessione del client
   HttpSession session = request.getSession(true);
   out = response.getWriter();
}
```

Il valore booleano passato alla `getSession()` serve per specificare la modalità con cui ottenere la sessione: nel caso di un `true`, il Servlet tenterà prima di ricavare la sessione se questa è già presente, altrimenti ne creerà una nuova. Il valore `false` invece permette di ottenere sessioni solo se queste sono già presenti.

Con la versione 2.1 della API è stato introdotto per comodità il metodo

```
request.getSession();
```

senza parametri, la cui invocazione ha lo stesso effetto di quella con parametro `true`.

Nel caso in cui la sessione venga creata ex novo, nel browser verrà memorizzato un cookie contenente l'identificativo univoco della sessione stessa. Per una sessione appena creata, il metodo `isNew()` restituirà `true`: in questo caso la sessione non contiene nessuna informazione.

Da notare che, sempre con la versione 2.1 della API l'interfaccia `HttpSessionContext` (che fornisce una interfaccia verso il contesto di sessione) è stata deprecata, così come il metodo `HttpSessionContext.getSessionContext()` che adesso restituisce una sessione nulla che non contiene informazioni.

Il motivo di questa scelta è legato alla necessità di garantire un livello maggiore di sicurezza, dato che un suo utilizzo permette a un Servlet di accedere a ed eventualmente modificare le informazioni memorizzate in una sessione creata a gestita da un altro Servlet.

Altra innovazione portata con la API 2.1 è la possibilità di impostare, oltre che con i tool forniti dal server, la durata massima della validità di una sessione per mezzo del metodo

```
public void setMaxInactiveInterval(int interval)
```

fornito direttamente dalla `HttpSession`, insieme all'anologo

```
public int getMaxInactiveInterval()
```

Come avviene con i Cookie, al fine di mantenere una sessione in modo corretto, si dovrà fare attenzione ad invocare la `getSession` prima di tutte le operazioni di scrittura verso il client, dato che questo blocca tutte la possibilità di agire sull'header e quindi di creare una sessione.

Una volta ottenuto l'oggetto `Session`, è possibile usarlo per memorizzare qualsiasi tipo di informazione relativo alla sessione logica che essa rappresenta, per mezzo degli opportuni metodi della `HttpSession`.

Ad esempio, per contare il numero di volte che un Servlet viene invocato da un certo determinato client, sarà sufficiente scrivere

```
// si prova a ricavare dalla sessione
// il valore di sessiontest.counter
Integer ival = (Integer) session.getValue("sessiontest.counter");

// se la sessione non esiste il valore restituito sarà null
if (ival==null)
   // ival non è definito
   ival = new Integer(1);
else {
   // il valore di ival è definito
   // si incrementa di 1
   ival = new Integer(ival.intValue() + 1);
}

// si memorizza nella sessione il nuovo valore
session.putValue("sessiontest.counter", ival);
```

Dunque `Session` è una specie di dizionario di valori a cui possiamo accedere da un qualsiasi Servlet lanciato dallo stesso client. Questa è un'altra importante differenza rispetto ai cookie, dove per ogni valore che si vuole memorizzare si deve installare un cookie apposito, con indubbio vantaggio sulla semplicità.

Cancellazione di una sessione

Una sessione utente può essere invalidata manualmente o, a seconda dell'ambiente su cui il Servlet è in esecuzione, automaticamente. Ad esempio il Java Web Server di Sun rimuove la sessione dopo un periodo di inattività del client a cui la sessione si riferisce di 30 minuti.

Invalidare una sessione significa eliminare l'oggetto di tipo `HttpSession` e quindi tutti i dati in esso memorizzati.

Una sessione rimane valida finché non viene invalidata esplicitamente mediante il metodo `invalidate()`, oppure supera il tempo massimo di inattività.

I motivi per cui una sessione debba essere eliminata, possono essere molteplici, come ad esempio la fine di un acquisto tramite web, nel caso in cui la sessione svolga la funzione di carrello della spesa virtuale.

Session tracking e disattivazione dei cookie

La gestione delle sessioni, pur essendo sicuramente uno strumento più potente del semplice cookie, fa sempre uso di tali oggetti: infatti, benché le informazioni siano memorizzate sul server, il `SessionID` viene memorizzato nel client per mezzo di un cookie.

Volendo permettere il corretto funzionamento di una applicazione basata su Servlet, indipendentemente dal fatto che il browser sia abilitato o meno al supporto per i cookie, si può ricorrere ad una tecnica detta URL rewriting: in questo caso il Servlet dovrà modificare la pagina HTML da inviare in risposta al client inserendo l'id della sessione come parametro per le invocazioni successive sul Servlet stesso

Ad esempio se il link al Servlet contenuto nella pagina fosse qualcosa del tipo

```
http://nome_host/servlet/nomeservlet
```

lo si dovrà trasformare in qualcosa del tipo

```
http://nome_host/servlet/nomeservlet$SessionID=234ADSADASZZZXXXSWEDWE$
```

Questa trasformazione può essere effettuata in modo del tutto automatico grazie ai metodi `encodeURL()` ed `encodeRedirectURL()` entrambi messi a disposizione dalla `HttpServletResponse`.

Più precisamente nel caso in cui si debba semplicemente codificare un URL con il session ID, si potrà utilizzare il primo, mentre nel caso di una redirect verso un'altra pagina si potrà fare uso del secondo.

L'utilizzo di tali metodi è piuttosto semplice: infatti, supponendo che il primo Servlet effettui una stampa riga per riga della pagina da inviare al client, il metodo `service()` potrebbe diventare

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response) throws  ServletException,
                                                        IOException {

// Ricava la sessione utente, il Writer, ecc.
...

// inizia la scrittura della pagina
out.println("<html>" + ...);
...

// scrive il codice HTML relativo a un link immettendo SessionID
out.print("<a href=");
out.print(response.encodeURL("/servlet/servlet2?p1="+p1+"));
out.println("> </a>");

// effettua la stampa di altro codice HTML
...

// scrive il codice HTML relativo a un secondo link
// immettendo il SessionID
out.print("<a href=");
out.print(response.encodeURL("/servlet/servlet2?p2="+p2+"));
out.println("> </a>");

...

// conclude la scrittura della pagina
out.println("</html>");

}
```

Da notare che il Servlet deve effettuare la modifica di tutti i link che verranno mostrati nella pagina HTML inviata al client.

Fatto molto interessante è che la ricodifica dell'URL avviene in maniera selettiva in funzione del fatto che il client abbia o meno abilitato il supporto per i cookie.

Da un punto di vista operativo, nel momento in cui l'utente effettua un click sul link modificato, il metodo `getSession()` del Servlet ricava automaticamente la sessione dall'URL, in modo analogo al caso in cui la sessione sia stata memorizzata nella cache del browser sotto forma di cookie.

Web Applications

Disponibili con la versione 2.2 della Servlet API le Web Application rappresentano forse una delle novità più interessanti introdotte da Sun, anche se non sono esclusivamente legate ai Servlet.

Una Web Application, come definita nella Servlet Specification, è una collezione di Servlet, Java Server Pages, classi Java di utilità, documenti statici come ad esempio pagine HTML, raggruppati all'interno di un archivio JAR in modo da semplificare al massimo le operazioni di installazione e configurazione.

Infatti, se in passato per effettuare il deploy e la configurazione di una applicazione basata su Servlet era necessario copiare una serie di file ed editarne altri, adesso con le Web Application è sufficiente copiare un unico file JAR in una directory opportuna del server: all'interno di tale file (che ha estensione `.war`) sono contenute tutte le risorse necessarie alla applicazione (`.class`, `.html`, immagini) ma anche tutte le informazioni per la configurazione dell'applicazione stessa. Ad esempio per poter mappare un Servlet con un nome logico e poterlo invocare da remoto, adesso è sufficiente editare un opportuno file XML dove sono indicati sia il nome del Servlet che i vari parametri di configurazione.

Chiunque abbia provato a utilizzare la vecchia modalità di installazione e configurazione si renderà conto di quanto questo possa essere comodo e vantaggioso.

Si limiterà comunque in questo capitolo la trattazione degli aspetti legati alla programmazione dei Servlet nell'ambito delle Web Application.

WAR Files

Il file JAR che contiene una Web Application in questo caso prende il nome di Web Application Archive (WAR file) e ha estensione `.war`.

Una tipica struttura di un file di questo tipo potrebbe essere ad esempio

```
index.htm
index.jsp
images/title.gif
images/mokalogo.gif
WEB-INF/web.xml
WEB-INF/lib/jspbean.jar
WEB-INF/classes/MyServlet.class
WEB-INF/classes/it/mokabyte/MokaServlet.class
```

Al momento della installazione, il file `.war` deve essere posizionato in una directory mappata poi dal server HTTP con un URI particolare. Tutte le richieste inoltrate a tale URI saranno poi gestite dalla applicazione contenuta in tale file WAR.

La directory `WEB-INF`

La directory `WEB-INF` ha un compito piuttosto particolare all'interno di una Web Application: i file qui contenuti infatti non sono accessibili direttamente dai client, ma sono utilizzati dal server per configurare l'applicazione stessa. Il suo funzionamento è quindi molto simile a quello della directory `META-INF` di un normale file JAR.

La sottodirectory `WEB-INF\classes` contiene tutti i file compilati dei vari Servlet e delle classi di supporto; la `WEB-INF\lib` invece contiene altre classi che sono però memorizzate in archivi JAR. Tutte le classi contenute in queste due directory sono caricate automaticamente dal server al momento del load della applicazione stessa.

I Servlet presenti nella dir `WEB-INF` possono essere invocati tramite un'invocazione che, nel caso dell'esempio appena visto, potrebbe essere

```
/nome_application/servlet/MyServlet
/nome_application/servlet/it.mokabyte.MokaServlet
```

Il file `web.xml` contenuto nella dir `WEB-INF` è noto come deployment descriptor file, e contiene tutte le informazioni relative all'applicazione in cui risiede. Si tratta di un file XML il cui DTD è fissato da Sun con una specifica che indica oltre 50 tag con i quali poter specificare una qualsiasi delle seguenti informazioni:

- le icone per la rappresentazione grafica dell'applicazione;
- la descrizione dell'applicazione;
- un flag che indica se la applicazione è *distributed* o no, ovvero se tale applicazione può essere condivisa fra diversi server remoti. I motivi per cui possa essere utile realizzare un'applicazione distribuita sono molti, dei quali forse il più importante è legato a una maggiore flessibilità e alla possibilità di implementare tecniche di balancing e carico computazionale ripartito dinamicamente. Le regole per progettare e scrivere un'applicazione distribuita sono molte ed esulano dallo scopo di questo capitolo, per cui si rimanda per maggiori approfondimenti alla bibliografia ufficiale;
- i parametri di inizializzazione della applicazione nel complesso o dei singoli Servlet;
- la registrazione del nome o dei nomi di un Servlet in esecuzione: possibilità questa che semplifica e soprattutto standardizza la fase di deploy di un Servlet;
- l'ordine di caricamento dei Servlet;
- le regole di mapping fra i vari Servlet ed i relativi URL;
- il timeout delle varie sessioni, opzione resa possibile grazie ai nuovi metodi di gestione delle sessioni introdotti con la API 2.2;
- il welcome file list, ovvero la sequenza dei file da utilizzare per la risposta da inviare al client (p.e.: `index.htm`, `index.html` o `welcome.htm`);

- le regole per la gestione degli errori, tramite le quali specificare fra l'altro anche quali pagine HTML (statiche o dinamiche) debbano essere visualizzate in corrispondenza del verificarsi di un errore del server (es. una pagina non trovata) o di una eccezione prodotta dal Servlet engine;

- i riferimenti aggiuntivi alle tabelle di lookup utilizzate durante i riferimenti con JNDI a risorse remote;

- le regole di policy, tramite le quali specificare vincoli aggiuntivi di sicurezza.

Da un punto di vista pratico, la struttura del file XML non è importante di per sé, ma piuttosto è importante il fatto che la sua presenza permette finalmente la possibilità di automatizzare e standardizzare il processo di installazione e deploy della applicazione così come di un gruppo di Servlet.

Grazie al meccanismo di installazione e deploy messo in atto con l'utilizzo dei file `.war`, si parla sempre più spesso di Pluggables Web Components, ovvero di componenti con funzionalità specifiche installabili in un application server in modo molto semplice e veloce.

Si pensi ad esempio alla possibilità di attivare un motore di ricerca semplicemente copiando un file in una directory opportuna, indipendentemente dal sistema operativo e dal server. Parallelamente, una azienda che desideri attivare servizi di hosting a pagamento, potrà utilizzare il sistema delle Web Applications e dei file `.war` per semplificare la gestione degli utenti e dei vari domini governati da Servlet.

La programmazione di una Web Application

Dal punto di vista della programmazione, una Web Application può essere vista come un `ServletContext`, di cui si è parlato in precedenza; tutti i Servlet all'interno della medesima Web Application condividono lo stesso `ServletContext`.

I parametri di inizializzazione specificati tramite il Deployment Descriptor File possono essere ricavati per mezzo dei metodi della interfaccia `ServletContext`

```
public String getInitParameter(String name)

public Enumeration getInitParameterNames()
```

che sono gli equivalenti dei metodi omonimi della classe `GenericServlet`.

Un Servlet può ricavare il prefisso dell'URL rispetto al quale è stato invocato tramite il nuovo metodo `getContextPath()` fornito dalla `ServletRequest`.

In questo caso viene restituita una stringa che rappresenta il prefisso dell'URL relativo al context in esecuzione: ad esempio per una richiesta a un Servlet in esecuzione in

```
/moka/articles/servlet/FinderServlet
```

il `getContextPath()` restituirà la seguente stringa

```
/moka/articles
```

Ogni Servlet conosce già il suo contesto per cui non è necessario includerlo tutte le volte che si vuole accedere alle risorse contenute nel contesto stesso se non si desidera dover effettuare una ricompilazione tutte le volte che si effettua una qualche modifica alla applicazione.

Java Server Pages

DI PAOLO AIELLO

Introduzione

Pagine web a contenuto dinamico

Il linguaggio HTML, grazie alla sua semplicità e al largo supporto ricevuto dai produttori di browser web, si è rapidamente affermato come uno standard per la pubblicazione di documenti su web. Nelle prime versioni del linguaggio i documenti HTML erano essenzialmente documenti *statici*. Pur avendo le notevoli capacità di un ipertesto di incorporare contenuti di varia natura e di stabilire collegamenti con altri documenti (link), le possibilità di interazione con l'utente restavano limitate. Per venire incontro all'esigenza di un maggiore interazione tra client e server e per fornire all'utente contenuti più vari e meno statici sono state via via introdotte svariate tecnologie, che introducono elementi *dinamici* nei siti web.

Il supporto di queste tecnologie ha comportato modifiche ai vari elementi coinvolti nell'interazione client–server:

- estensione delle funzionalità del client (web browser)
- estensione delle funzionalità del server
- estensione del linguaggio HTML.

Le estensioni delle funzionalità del client e del server sono state ottenute sia con la realizzazione di nuove versioni dei browser e dei server, sia con la realizzazione di moduli aggiuntivi, detti *plugin*, che vengono collegati, tramite opportune API, con il browser o

con il server web. Nel caso dei server, questi plugin sono chiamati anche *Application Server* (ma è un termine ambiguo usato anche per indicare server complessi che fungono da web server, da transaction server, da server per oggetti distribuiti ecc.) o *Server Extension*.

Anche se generalmente queste tecnologie comportano qualche supporto da parte del client, del server ma anche del linguaggio, è possibile stilare una classificazione sulla base di *dove viene eseguito il codice* che compie le operazioni e le elaborazioni che rendono il funzionamento *dinamico*. In base a tale criterio si hanno le seguenti categorie.

Tecnologie lato client (*client side*)

Nelle tecnologie lato client la dinamicità e l'interazione con l'utente sono gestite direttamente da codice eseguito dal client o da un suo plugin. È il caso degli script supportati dai browser (JavaScript, VBScript, ecc.) e delle Applet Java, che sono diventati elementi standard di una pagina web, con uno specifico supporto nel linguaggio HTML (interazione tra oggetti HTML e script, tag `APPLET`).

Tecnologie lato server (*server side*)

Nelle tecnologie lato server, il client ha un ruolo essenzialmente passivo, ed è il server, o un suo plugin, a gestire la parte dinamica. Il caso più tipico è la comunicazione client–server per mezzo del protocollo HTTP, supportata in HTML dai *form* e dai suoi componenti. Attraverso questi componenti l'utente può inserire dei dati, mandarli al server e ottenerne una risposta. Il browser manda questi dati sotto forma di una *richiesta HTTP*, utilizzando uno dei *metodi* previsti dal protocollo (in genere `get` o `post`) a cui il server fa seguire una appropriata risposta. Naturalmente, perché il server sia in grado di elaborare la risposta, è necessario che supporti tecnologie adeguate. Poiché si ha a che fare non con richieste e risposte predefinite, ma stabilite invece dagli sviluppatori di volta in volta, queste tecnologie devono innanzitutto fornire al programmatore i mezzi per scrivere sul lato server il codice che elabora le risposte (sul lato client il supporto per l'invio delle richieste è fornito dai form HTML). In queste categoria troviamo CGI, le Active Server Pages (ASP) di Microsoft e, in Java, i Servlet e le Java Server Pages.

Approcci misti

Presentano elementi dinamici sia sul lato server che sul lato client. Spesso queste tecnologie sono basate sull'uso di plugin sia sul lato client che su quello server che interagiscono direttamente tra loro (scavalcando eventualmente il normale canale di comunicazione client–server, basato su HTTP) e si servono del client e del server solo come di una sorta di "punto d'appoggio" per il loro funzionamento. Rientrano in questo caso l'uso congiunto di Applet e Servlet Java come anche diversi tipi di estensioni multimediali per il web,

tipo Real Player™, Macromedia Flash™, ecc., le quali consentono al server di inviare, e al client di ricevere, suoni e animazioni, ma soprattutto consentono al client di presentare all'utente questi contenuti attraverso specifici plugin che visualizzano l'animazione o leggono un file di suoni e lo inviano, opportunamente elaborato, alla scheda sonora.

Tecnologie server side

Tra questi diversi approcci, assume particolare rilevanza nell'ambito delle applicazioni business, quello delle tecnologie *server side*, basate sul protocollo HTTP. Sono infatti quelle che più si prestano allo sviluppo di applicazioni web centrate sulla gestione di basi di dati. Si descrive brevemente il funzionamento delle tecnologie più diffuse, alle quali si è già fatto cenno.

CGI (Common Gateway Interface)

È un'interfaccia per mezzo della quale il server web è in grado di lanciare delle applicazioni che risiedono sul server, in una particolare locazione riservata. Le applicazioni sono normali file eseguibili tramite una shell che disponga di standard I/O e di variabili d'ambiente, elementi entrambi utilizzati per lo scambio di dati tra il server e l'applicazione CGI. Quest'ultima può essere sia un eseguibile binario che uno script interpretato direttamente dalla shell o da un altro interprete (i linguaggi più usati in questo contesto sono Perl, C e Python). Tutto ciò che l'applicazione scrive sullo standard output viene intercettato dal server e inviato al client. Il principale svantaggio di questa tecnica è il consumo di risorse del sistema, dato che ogni esecuzione CGI dà origine a un nuovo processo.

ASP (Active Server Pages)

È una tecnologia sviluppata da Microsoft a supporto di altre sue tecnologie proprietarie, principalmente gli oggetti ActiveX e il linguaggio VBScript, simile al VisualBasic (ma ASP supporta anche JavaScript). L'idea base di ASP è di dare la possibilità di inserire contenuti dinamici in una pagina web inframmezzando il codice HTML con codice in linguaggio script che viene eseguito dal server. Questa esecuzione ha come effetto sia il compimento di operazioni particolari nell'ambiente server (ad esempio la modifica di un database), sia eventualmente l'invio di dati al client (in genere in formato HTML), che vengono inseriti nella pagina ASP al posto degli script. Gli script, da parte loro, possono sia eseguire direttamente delle operazioni, sia utilizzare oggetti ActiveX residenti sul server.

Servlet e JSP

Le tecnologie Java, nate per ultime, hanno usufruito delle precedenti idee ed esperien-

ze, apportando però anche significativi miglioramenti. I Servlet sono la versione Java di CGI, con i grossi vantaggi di un minore impiego di risorse, dato che tutti i Servlet sono eseguiti dalla stessa Java virtual machine, messa in esecuzione dal server, come thread distinti e non come processi distinti. Si hanno poi a disposizione tutta la potenza e l'eleganza del linguaggio Java e delle sue innumerevoli librerie standard ed estensioni standard.

JSP invece si ispira direttamente ad ASP, con somiglianze notevoli anche nella sintassi e nell'impostazione generale. Rispetto alle tecnologie Microsoft, oltre ai vantaggi generici legati all'uso di Java che abbiamo già menzionato per i Servlet, si ha quello di una reale portabilità in tutte le piattaforme su cui esista un'implementazione della macchina virtuale Java (ossia tutte quelle comunemente usate).

Generazione dinamica di pagine web

Quello che accomuna le diverse tecnologie server side, è che in tutti i casi si ha la generazione di pagine web (cioè pagine HTML) come risultato di una determinata richiesta da parte del client. La pagina non esiste come risorsa statica, ma viene generata dinamicamente.

Nel caso di CGI e Servlet, la pagina viene interamente costruita dal codice dell'applicazione, che scrive nello stream di output il codice HTML, riga per riga. Naturalmente, salvo casi estremamente semplici, chiamare funzioni o metodi di output che scrivano letteralmente il codice HTML risulta un'operazione rudimentale, decisamente poco elegante e portatrice di continui errori. Insomma, niente che possa essere raccomandabile per un programmatore.

Per ovviare a questo inconveniente, in ambito Java, sono stati escogitati diversi sistemi:

- librerie di funzioni o di oggetti (a seconda del linguaggio) che rappresentano i diversi tag HTML. In tal modo gli attributi HTML diventano argomenti delle funzioni che restituiscono la stringa che definisce l'elemento HTML. Si rende così il codice più ordinato e leggibile e si evitano errori di sintassi nella generazione del codice HTML.

- tools e librerie che operano come processori di macro o, se si vuole, *template processor*, che sostituiscono, nelle pagine HTML, speciali tag predefiniti o definiti dall'utente con altri contenuti anch'essi predefiniti o definiti dall'utente. Questi tools operano generalmente senza alcun collegamento con il web server e con l'application server e forniscono delle classi Java (in genere dei Servlet) che possono poi essere utilizzate con un application server.

- le Java Server Pages, il cui engine opera in modo simile ai template processor, ma è integrato in un application server. L'intero processo di generazione del codice, com-

pilazione ed esecuzione è gestito automaticamente in modo trasparente. Un'altra peculiarità di JSP è che permette l'inserimento diretto di codice Java nella pagina. JSP è inoltre l'unico sistema *standard*, supportato direttamente da Sun.

Che cosa è una Java Server Page

Ora che JSP è stato inquadrato nel suo contesto ed è stata illustrata a grandi linee la sua ragion d'essere, si prenderà in considerazione questa tecnologia da un punto di vista più strettamente tecnico.

Da un punto di vista funzionale una pagina JSP si può considerare come un file di testo scritto secondo le regole di un *markup language* in base al quale il contenuto del file viene elaborato da un *JSP container* (così viene chiamato un software per l'elaborazione di pagine JSP), per restituire il risultato di una trasformazione del testo originale, secondo le istruzioni inserite nel testo. Si tratta quindi *di pagine web a contenuto dinamico* generato al momento in cui la pagina viene richiesta dal client.

A differenza di interpreti di linguaggi come XML e HTML, un JSP container si comporta più come un *template processor* (o pre-processore, sul modello di quello del linguaggio C) che come un semplice markup processor. Infatti una pagina JSP è un file in formato testo che comprende essenzialmente due tipi di testo:

- *template text* ovvero testo "letterale" destinato a rimanere tale e quale dopo l'elaborazione della pagina;

- *JSP text* porzioni di testo che vengono interpretate ed elaborate dal JSP container.

Quindi in JSP solo alcune parti del testo vengono interpretate ed elaborate, mentre non ci sono restrizioni sul contenuto del template text. In pratica, nella maggior parte dei casi, il template text è in formato HTML (o a volte XML), dato che le JSP sono pagine web.

Dal punto di vista del programmatore Java, invece, una Java Server Page può essere vista come un modo particolare per interfacciarsi a oggetti Java, in particolare a Servlet e bean. Infatti il JSP container converte la pagina JSP in un Servlet, generando prima il codice sorgente, poi compilandolo. Il Servlet così generato potrà interagire con componenti JavaBeans, secondo le istruzioni espressamente inserite nella pagina JSP. Per capire meglio come questo avvenga, consideriamo il seguente esempio.

Un semplice esempio

Questo è probabilmente l'esempio più semplice che si possa fare di una pagina JSP.

```
<html>
<head>
<title>Data e ora</title>
</head>
<body>
<p>
Data e ora corrente:<br>
<b><%= new java.util.Date() %></b>
</p>
</body>
</html>
```

Si tratta, come si vede, di una normale pagina HTML, con un solo elemento estraneo a questo linguaggio:

```
<%= new java.util.Date() %>
```

Si tratta di una *espressione* JSP, in linguaggio Java, che verrà interpretata e valutata dal JSP container, e sostituita dalla data e ora corrente, anche se in un formato standard che non è probabilmente molto utile.

Questo significa che l'espressione `new java.util.Date()` è stata eseguita e convertita in stringa.

Elaborazione di una pagina: JSP e Servlet

Ma, in definitiva, in quale maniera la pagina è stata elaborata dal JSP container? In primo luogo è stato generato il codice che definisce una classe Java; come si diceva, questa classe è generalmente un Servlet. Il codice generato potrebbe assomigliare al seguente.

```
class JSPRawDate extends HttpJspServlet {
   public void _jspService (HttpServletRequest request,
                            HttpServletResponse response) {
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println();
      out.println("<head>");
      out.println("<title>Data e ora</title>");
      out.println("</head>");
      out.println();
      out.println("<body>");
      out.println("<p>");
      out.println("Data e ora corrente:<br>");
      out.print("<b>")
      out.print(new java.util.Date());
      out.println("</b>");
      out.println("</p>");
```

```
        out.println("</body>");
        out.println();
        out.println("</html>");
    }
}
```

La classe è derivata da una ipotetica classe `HttpJspServlet`, che potrà essere una sottoclasse di `HttpServlet` e *dovrà* implementare l'interfaccia `HttpJspPage`, definita nella JSP API. Quest'interfaccia contiene il metodo `_jspService()`, corrispondente al metodo `service()` del Servlet generico.

Come si vede, il Servlet non fa altro che rimandare in output le parti del file che non contengono tag JSP come `string literal` mentre l'espressione JSP viene eseguita e valutata prima di essere scritta nello stream. Naturalmente ci si può aspettare che il codice effettivamente generato da un JSP container sia parecchio diverso e meno immediatamente comprensibile di questo, dal momento che probabilmente ci sono mezzi più efficienti per copiare il testo nello stream di output e che, inoltre, non sono stati presi in considerazione alcuni aspetti, come la "bufferizzazione" dell'output, di cui si parlerà in seguito; e anche il nome della classe probabilmente sarà diverso. Tuttavia, nella sostanza, quello che il codice farà sarà equivalente.

Dopo la generazione del codice Java, è necessario compilare il codice per ottenere il bytecode eseguibile dalla virtual machine. Il processo di compilazione sarà lanciato una sola volta, alla prima richiesta, dopodiché sarà utilizzata la classe Java già compilata tutte le altre volte. In sostanza si può pensare a JSP come a un modo per generare automaticamente un Servlet ad hoc per la gestione di una pagina web a contenuto dinamico, ossia come a una forma di *Servlet authoring*.

In effetti le specifiche JSP prevedono esplicitamente la possibilità di distribuire pagine JSP precompilate sotto forma di servlet (bytecode Java) e forniscono anche indicazioni sulle modalità di packaging, ossia di "confezionamento" per la distribuzione. Questa forma di distribuzione ha il vantaggio di evitare l'attesa iniziale dovuta alla compilazione e quello di non richiedere la presenza di un Java compiler sulla macchina server.

Quindi, prescindendo dai dettagli implementativi, l'esecuzione di una pagina JSP prevede due fasi distinte:

- fase di *traduzione–compilazione*, durante la quale viene costruito un oggetto eseguibile dalla VM (in genere un Servlet) in grado di elaborare una risposta alle richieste implicite o esplicite contenute nella pagina;

- fase di *processing*, in cui viene mandato in esecuzione il codice generato e viene effettivamente elaborata e restituita la risposta.

Gli elementi che compongono una Java Server Page

Finora si sono visti solo alcuni degli elementi di una pagina JSP: il template text e le espressioni. Di seguito si fornisce un elenco completo di tali elementi, con una breve descrizione di ciascuno.

Template text

Tutte le parti di testo che non sono definite come elementi JSP; vengono copiate tali e quali nella pagina di risposta.

Comment

Con sintassi: `<%-- comment --%>`; sono commenti che riguardano la pagina JSP in quanto tale, e pertanto vengono eliminati dal JSP container nella fase di traduzione–compilazione; da non confondere con i commenti HTML e XML, che vengono inclusi nella risposta come normale template text.

Directive

Con sintassi: `<%@ directive ... %>`; sono direttive di carattere generale, indipendenti dal contenuto specifico della pagina, relative alla fase di traduzione–compilazione.

Action

Seguono la sintassi dei tag XML ossia `<tag attributes> body </tag>` oppure `<tag attributes/>` dove `attributes` sono nella forma `attr1 = value1 attr2 = value2 ...`; le azioni sono eseguite nella fase di processing, e danno origine a codice Java specifico per la loro esecuzione.

Scripting element

Sono porzioni di codice in uno *scripting language* specificato nelle direttive; il linguaggio di default è lo stesso Java. Si dividono in tre sottotipi:

Scriptlet
Con sintassi `<% code %>`; sono porzioni di codice nello scripting language che danno

origine a porzioni di codice Java; generalmente inserite nel metodo `service()` del Servlet; se contengono dichiarazioni di variabili queste saranno variabili locali valide solo nell'ambito di una singola esecuzione del Servlet; se il codice contiene istruzioni che scrivono sullo stream di output, il contenuto mandato in output sarà inserito nella pagina di risposta nella stessa posizione in cui si trova lo scriptlet.

Declaration

Con sintassi `<%! declaration [declaration] ... %>`; sono dichiarazioni che vengono inserite nel Servlet come elementi della classe, al di fuori di qualunque metodo. Possono essere sia variabili di classe che metodi. Se si tratta di variabili, la loro durata è quella del Servlet stesso; di conseguenza sopravvivono e conservano il loro valore nel corso di tutte le esecuzioni dello stesso oggetto Servlet.

Expression

Con sintassi `<%= expression %>`; contengono un'espressione che segue le regole delle espressioni dello scripting language; l'espressione viene valutata e scritta nella pagina di risposta nella posizione corrispondente a quella dell'espressione JSP.

Nelle prossime sezioni si esamineranno meglio i vari elementi, cominciando con gli scripting element.

Scripting element

Linguaggi di script

Come accennato precedentemente, il linguaggio di script non deve essere necessariamente Java, anche se è richiesto dalle specifiche che un JSP container supporti il linguaggio Java come linguaggio di default. Un container può supportare, oltre a Java, qualunque altro linguaggio di script, alle seguenti condizioni:

- il linguaggio deve essere in grado di manipolare oggetti Java e chiamare i loro metodi;
- deve esserci un supporto per le exception Java.

Nel seguito di questa trattazione, per semplificare, l'ipotesi considerata sarà che il linguaggio di script sia Java.

Oggetti impliciti

Si è visto come esiste una corrispondenza uno a uno tra una pagina JSP e il Servlet nel quale viene tradotta. Inoltre l'esecuzione della pagina JSP corrisponde all'esecuzione del metodo `service()` del Servlet. Come è noto, il Servlet manipola una serie di oggetti per

svolgere il suo lavoro, principalmente i due oggetti `ServletRequest` e `ServletResponse`, su cui si basa tutto il meccanismo di funzionamento. Quindi, per poter usufruire dei servizi del Servlet nella pagina JSP, occorre avere un modo per accedere a questi oggetti: a questo scopo esistono gli *oggetti impliciti* JSP, utilizzabili in tutti gli scriptlet. Eccone un elenco.

`Request`

Corrisponde generalmente all'oggetto `ServletRequest` passato come parametro al metodo `service()` del Servlet; può essere una qualunque sottoclasse di `ServletRequest`; generalmente si tratta di una sottoclasse di `HttpServletRequest`.

`Response`

Corrisponde all'oggetto `ServletResponse` passato come parametro al metodo `service()` del Servlet; può essere una qualunque sottoclasse di `ServletResponse`; generalmente si tratta di una sottoclasse di `HttpServletResponse`.

`Out`

Poiché il container deve fornire un meccanismo di buffering, non è dato accesso direttamente all'oggetto `PrintWriter` restituito da `response.getWriter()`. L'oggetto `out` è invece uno stream bufferizzato di tipo `javax.servlet.jsp.JspWriter`, restituito da un metodo del `PageContext`. Il trasferimento sullo output stream del `ServletResponse` avviene in un secondo tempo, dopo che tutti i dati sono stati scritti sull'oggetto `out`.

`Page`

È un riferimento all'oggetto che gestisce la richiesta corrente, che si è vista essere generalmente un Servlet. In Java corrisponde al `this` dell'oggetto, pertanto è di scarsa utilità. Si presume risulti utile con altri linguaggi di script.

`PageContext`

Si tratta di un oggetto della classe `javax.servlet.jsp.PageContext` utilizzata pre-

valentemente per incapsulare oggetti e features particolari di ciascuna implementazione del container. Il `PageContext` contiene ad esempio un metodo che restituisce l'oggetto `out`, altri che restituiscono gli oggetti `session`, `application`, e così via. Un particolare container potrebbe ad esempio restituire come oggetto `out` una istanza di una speciale sottoclasse di `JspWriter`, o restituire tipi di oggetti diversi a seconda di altre informazioni di contesto. Tutto questo sarebbe nascosto dall'interfaccia di `pageContext` e la pagina, anche in formato precompilato, potrebbe funzionare senza problemi in qualunque altro JSP container. Il `PageContext` viene utilizzato anche per condividere oggetti tra diversi elementi, come sarà descritto in seguito (si veda oltre “Tag extension”, “Cooperazione tra azioni”).

`Session`

Corrisponde all'oggetto `HttpSession` del Servlet, viene restituito da un metodo del `PageContext`.

`Application`

Corrisponde al `ServletContext` del Servlet, viene restituito da un metodo del `PageContext`.

`Config`

Corrisponde al `ServletConfig` del Servlet, viene restituito da un metodo del `PageContext`.

`Exception`

Quest'oggetto è disponibile solo all'interno di una error page (si veda “Gestione degli errori” per dettagli).

Una *tag extension library* (vedere “Tag extension”) può definire altri oggetti impliciti. I nomi che cominciano con `jsp`, `_jsp`, `jspx` e `_jspx` sono riservati.

Qualche esempio

Il seguente esempio riprende quello precedente, ma questa volta la data e l'ora vengono mostrate separatamente e opportunamente formattate.

```
<html>
<head>
   <title>Data e ora</title>
</head>
<%@ page import = "java.util.*, java.text.*" %>
<%!
String DateFormat = "EEEE d MMMM yyyy";
   DateFormat dateFormat = new SimpleDateFormat(DateFormat);
   DateFormat timeFormat = new SimpleDateFormat("H:mm");
%>
<% Date dateTime = new Date(); %>
<body style = "font-size: 16pt;">
Oggi &egrave; <%= dateFormat.format(dateTime) %><br>
e sono le ore <%= timeFormat.format(dateTime) %>
</body>
</html>
<html>
```

Il primo elemento JSP è una *direttiva page* di cui si specifica l'attributo `import` per importare dei package Java (si vedano più avanti i paragrafi dedicati alle direttive). Poi c'è una *JSP declaration* con cui si creano due oggetti di tipo `DateFormat`. Si usano le dichiarazioni JSP perché si desidera creare questi oggetti solo all'inizio e non a ogni esecuzione, dato che il formato della data e dell'ora restano costanti. Si usa invece uno *scriptlet* di una sola riga per creare un oggetto di tipo `Date`. Poiché il valore della data e dell'ora corrente viene impostato al momento della creazione, è necessario creare un'istanza a ogni esecuzione. Gli ultimi due elementi sono delle espressioni JSP con le quali si visualizzano data e ora.

Ecco ora un esempio in cui si fa uso dell'oggetto implicito `request` e di istruzioni condizionali.

```
<html>
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 20pt;">
<% String name = request.getParameter("name"); %>
<% if (name == null || name.length() == 0) { %>
Ciao, chiunque tu sia!
<% } else { %>
Ciao <%= name %>!
<% } %>
</body>
</html>
```

L'oggetto `request` viene utilizzato per prelevare il valore del parametro `name`, ricevuto tramite il `post` di un form HTML. Il contenuto del messaggio visualizzato dipende dal fatto che questo parametro esista e non sia una stringa vuota.

Le istruzioni condizionali sono inserite in piccoli scriptlet Java; come si vede, il codice può essere frammentato liberamente e inframezzato da altri elementi. In questo modo la dinamicità è realizzata in modo abbastanza semplice ed immediato, ma il codice non eccelle in leggibilità e pulizia.

Questa è una pagina HTML da cui viene chiamata la pagina JSP.

```
<html>
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 16pt;">
<form action = "/examples/jsp/hello/hello.jsp" method = "post">
Scrivi qui il tuo nome
<input type = "text" name = "name">
<input type = "submit" value = "Clicca qui">
</form>
</body>
</html>
```

Action element

Action e script

Come gli script, le action sono istruzioni che vengono eseguite in fase di processing, ossia nel momento in cui viene elaborata la risposta, e si traducono in istruzioni Java nel metodo `service()` del Servlet. Ma, a differenza degli script, seguono la sintassi XML, cioè sono in pratica dei tag XML, e sono quindi indipendenti da qualunque linguaggio di script. Anche se la sintassi XML è piuttosto prolissa e non particolarmente elegante, ha il vantaggio di essere piuttosto semplice, e quindi più facilmente alla portata di un web designer che spesso non ha competenze specifiche di programmazione. La specifiche JSP definiscono una serie di tag standard e un meccanismo per la definizione di nuovi tag (le cosiddette *tag extension libraries* per le quali si rimanda più oltre a *Tag extension*).

Le action, in quanto elementi XML, possono essere del tipo dotato di un *body*, ossia con uno *start-tag* e un corrispondente *end-tag*, oppure del tipo *empty-element* con un unico tag, ed eventualmente uno stesso elemento può assumere tutte e due le forme.

I valori degli attributi sono normalmente rappresentati da elementi statici, ossia da valori esplicitamente specificati da una stringa di testo, valutati nella fase di traduzione; ossia si tratta di *translation-time values*. Ci sono tuttavia alcuni attributi il cui valore può essere rappresentato da espressioni JSP, assumendo la forma

```
<tagName attribute = "<%= expression %>" >
```

In questo caso si parla di *request-time values*, cioè di attributi che vengono valutati in fase di processing della richiesta. In questi casi il valore deve essere indicato soltanto da

un'unica JSP expression, che non può essere mischiata con altro testo o altre espressioni. Gli attributi delle standard action che accettano valori in questa forma sono esplicitamente indicati dalle specifiche.

Standard action

Le specifiche prevedono che le azioni standard siano implementate in tutti i container JSP, lasciando la possibilità di definire altri tags non standard per ciascuna implementazione. Tutti i tag delle azioni standard usano il namespace XML `jsp`, che è riservato e non può essere usato per le tag extension. Le azioni standard si possono dividere in due categorie:

- action per l'uso di componenti JavaBeans;
- altre action per compiere varie operazioni su pagine JSP, in fase di processing.

JSP e JavaBeans

I componenti JavaBeans sono probabilmente il mezzo migliore, più pulito ed elegante, per inserire contenuti dinamici in una pagina JSP. Mentre gli script finiscono con lo "sporcare" una pagina JSP mischiando tag HTML o XML e codice del linguaggio script, i bean, interfacciati per mezzo delle action, si integrano benissimo nel contesto della pagina che in tal modo contiene solo tag in un formato familiare a tutti i creatori di pagine web. In questa maniera si ha la massima separazione del codice dalla presentazione e di conseguenza la massima manutenibilità. I tag per la manipolazione di bean sono tre:

- `jsp:useBean`: serve per utilizzare un bean già esistente o creare una nuova istanza di un bean.
- `jsp:getProperty`: inserisce nella pagina il valore di una proprietà del bean.
- `jsp:setProperty`: assegna il valore di una o più proprietà del bean.

Nell'attuale versione di JSP (1.1) queste azioni non possono essere usate con gli Enterprise Java Beans. Questi possono essere manipolati solo attraverso script o altri Bean.

Prima di descrivere nei dettagli queste azioni e i loro attributi, si riporta di seguito un semplice esempio.

Si tratta della versione "Bean" dell'esempio "Data e ora" precedentemente realizzato con gli script element.

```
<html>
<head>
   <title>Data e ora</title>
</head>
<jsp:useBean id = "dateTime" class = "dates.DateTime" />
<body style = "font-size: 16pt;">
Oggi è <jsp:getProperty name = "dateTime" property
= "date"/><br> e sono le ore <jsp:getProperty name = "dateTime" property = "time"/>
</body>
</html>
<html>
```

Nell'azione `jsp:useBean` si specifica il nome dell'oggetto (`dateTime`) e quello della classe (la classe `DateTime` del package `dates`), mentre nelle azioni `jsp:getProperty` specifichiamo sempre il nome dell'oggetto e quello della proprietà (`date` e `time` rispettivamente).

Anche in un esempio così semplice si può apprezzare la differenza tra la versione script e quella bean: la pagina si presenta molto più pulita e compatta e il contenuto risulta di facile e intuitiva comprensione. Tutto il codice che prima era sparso per la pagina è ora incapsulato in una classe Java che segue le specifiche dei JavaBeans.

```
import java.util.*;
import java.text.*;
public class DateTime {
   DateFormat dateFormat = new SimpleDateFormat("EEEE d MMMM yyyy");
   DateFormat timeFormat = new SimpleDateFormat("H:mm");
   public String getDate() {
      Date date = new Date();
      return dateFormat.format(date);
   }

   public String getTime() {
      Date date = new Date();
      return timeFormat.format(date);
   }
}
```

L'azione `jsp:useBean`

Come si è già detto, questo tag serve per localizzare — si spiegherà meglio fra poco cosa si intende — un bean già esistente oppure per creare un nuovo bean del tipo specificato. Questi sono gli attributi dell'action:

`id`

Definisce una variabile che identifica il bean nello *scope* specificato con l'omonimo attributo. Questo identificatore è *case sensitive* e può essere utilizzato in espressioni o scriptlet.

`scope`

Definisce l'ambito di esistenza e di visibilità della variabile `id`. Il valore di default è `page`. Questi sono i possibili valori di `scope`:

- `page`: la variabile `id` è utilizzabile solo all'interno della pagina in cui compare il tag, o di una pagina inclusa staticamente (vedi la direttiva `include`).
- `request`: la variabile id è utilizzabile nell'ambito di una singola *request*. Come si vedrà, una pagina JSP può inoltrare la richiesta ricevuta a un'altra pagina JSP o a un Servlet, per cui non sempre lo scope `request` corrisponde allo scope `page`.
- `session`: la variabile è utilizzabile nell'ambito di un'intera sessione. Il bean esiste per tutta la durata della sessione, e ogni pagina che usa la sessione può accedervi. La pagina in cui il bean è creato deve contenere una direttiva `page` con l'attributo `session = true`.
- `application`: la variabile è utilizzabile nell'ambito di un'intera applicazione JSP, da tutte le pagine dell'applicazione.

`class`

Questo attributo viene utilizzato per identificare la classe a cui il bean appartiene, con eventuali specificatori del package. Sia il nome del package che quello della classe sono *case sensitive*. Se il bean viene creato perché non esistente, questa è la classe di cui sarà creata un'istanza. Di conseguenza, questo attributo deve essere presente se si vuole che il bean venga creato all'occorrenza, e deve indicare una classe non astratta provvista di un costruttore pubblico senza argomenti.

`type`

Identifica il tipo della variabile `id`, che rappresenta il bean localmente, ossia nella pagi-

na JSP. Questo tipo non deve necessariamente corrispondere a quello indicato dall'attributo `class`, ma può essere anche una superclasse della classe di appartenenza o un'interfaccia implementata dalla classe. Se `type` viene omesso, `id` sarà del tipo effettivo del bean. L'attributo `type` non viene usato per la creazione di un nuovo oggetto, per la quale è necessario comunque specificare l'attributo `class`. L'identificatore di tipo, come quello di classe, è case sensitive.

`beanName`

L'attributo `beanName` può essere usato al posto dell'attributo `class`, per specificare il tipo di bean che si vuole creare. Se si usa `beanName` il bean viene creato per mezzo del metodo `java.beans.Beans.instantiate()` che controlla prima se la classe è presente in forma serializzata (in un file con un nome del tipo `PackageName.ClassName.ser`) e, nel caso, lo crea a partile dal file, usando un `ClassLoader` (vedere il capitolo sui Beans per maggiori informazioni). L'attributo `beanName` può ricevere un valore di tipo *request-time* value, cioè gli si può assegnare un'espressione JSP che verrà valutata al momento dell'elaborazione della richiesta. Si descrivono ora più in dettaglio le operazioni con cui si svolge l'esecuzione di questa azione.

Se viene trovato un oggetto di nome corrispondente a quello specificato nell'attributo `id` nello scope specificato, viene creata una variabile locale con lo stesso nome e ad essa viene assegnato il reference dell'oggetto.

Se l'oggetto non viene trovato, il container cercherà di crearlo; perché l'operazione abbia successo è necessario che il container trovi un file `.class` corrispondente nel suo classpath. Se viene specificato l'attributo `beanName` piuttosto che l'attributo `class`, l'oggetto potrà essere creato anche a partire da una forma serializzata. Come nel caso precedente, il reference dell'oggetto viene assegnato a una variabile locale.

Se non è possibile né trovare il bean nello scope specificato, né crearne un'istanza, viene generata una `InstantiationException`.

L'azione `jsp:useBean` può essere usata nelle due forme, con start-tag ed end-tag, oppure con un solo empty-element-tag. Nel primo caso tutto quello che sta all'interno del body sarà eseguito soltanto nel caso che il bean venga creato; se invece il bean è già esistente tutto il contenuto del body viene ignorato.

Scope e oggetti impliciti

Si sarà notato che esiste una stretta corrispondenza tra i valori di scope e alcuni degli

oggetti impliciti disponibili in uno scriptlet. Questo non è un caso e in effetti lo scope di un bean è determinato dalla sua appartenenza a uno di questi oggetti impliciti, dato che l'oggetto implicito ne incapsula un riferimento. Se lo scope è la pagina, il bean sarà creato e rilasciato (cioè lasciato a disposizione del garbage collector) direttamente nel metodo `service()` del Servlet. Altrimenti in questo metodo sarà dichiarata comunque una variabile del tipo indicato dall'attributo `type`, ma l'oggetto referenziato sarà assegnato come attributo all'oggetto indicato dall'attributo `scope`.

In realtà chi si fa carico di mantenere e rendere disponibile il bean nello scope specificato è il `PageContext`, ma ci si può aspettare che questo a sua volta si serva degli oggetti `request`, `session` e `application`. Ecco un esempio. Si supponga che in una pagina JSP sia definita le seguente azione:

```
<jsp:useBean id = "bean" class = "SpecializedBean" type
= "BaseBean" scope = "session" />
```

Nel metodo `_jspService()` del Servlet si troverà un codice di questo genere:

```
BaseBean bean = (BaseBean)pageContext.getAttribute("bean", PageContext.SESSION_SCOPE);
if (bean == null)
   bean = new SpecializedBean();
pageContext.setAttribute("bean", bean, PageContext.SESSION_SCOPE)
```

Prima si controlla se l'oggetto è già presente nello scope specificato, per mezzo del metodo `PageContext.getAttribute()`. Se non è presente, viene creato e assegnato allo scope, con il metodo `setAttribute()`. Notare che la variabile dichiarata è del tipo specificato dall'attributo *type*, ossia `BaseBean` mentre il tipo effettivo dell'oggetto creato è quello indicato dall'attributo *class*, cioè `SpecializedBean`. L'oggetto `pageContext` incapsula l'intera operazione, ma ci si può aspettare che al suo interno assegni a sua volta il bean all'oggetto `session` tramite il metodo `HttpSession.setValue()`.

Quindi, all'interno di uno scriptlet, sarà possibile accedere ai bean anche attraverso gli oggetti impliciti, a seconda dello scope:

- con una chiamata del tipo `(BeanType)request.getAttribute("beanName")` per lo scope `request`;
- con una chiamata del tipo `(BeanType)session.getValue("beanName")` per lo scope `session`;
- con una chiamata del tipo `(BeanType)application.getAttribute("beanName")` per lo scope `application`;

oppure si potrà usare il metodo `pageContext.getAttribute()`, come nell'esempio.

In questo modo non sarà necessario usare un'azione `jsp:useBean`.

L'azione `jsp:getProperty`

Con tale azione si può ottenere il valore di una proprietà dell'oggetto specificato. Il bean deve essere reso accessibile con una precedente azione `jsp:UseBean`. Gli attributi sono:

`name`

Il nome del bean, corrispondente all'attributo `id` di `jsp:useBean`, oppure a quello di uno degli oggetti impliciti.

`property`

Il nome della proprietà; secondo le regole dei JavaBean, possono essere considerate proprietà del bean tutte quelle per cui esiste una coppia di metodi `getxxx() setxxx()`, e il nome della proprietà comincia sempre con una lettera minuscola (cioè, ad esempio, il metodo `getName()` dà origine alla proprietà `name`).

Se l'oggetto non esiste (quindi non è stato creato con `jsp:useBean`) viene generata un'eccezione. Le specifiche non dicono cosa succede se la proprietà ha valore `null`. Nell'implementazione di riferimento Sun, viene generata una `nullPointerException`.

L'azione `jsp:getProperty` non è utilizzabile con proprietà indicizzate. In alcuni testi non aggiornati si trovano esempi di uso di `jsp:getProperty` con oggetti impliciti (ad esempio `request`). Le specifiche non prevedono quest'uso del tag, anche se è possibile che funzioni in alcune implementazioni di JSP container.

L'azione `jsp:setProperty`

Il funzionamento di questa azione è un po' più complesso della precedente. Infatti questa azione è implicitamente legata all'oggetto `request` e in particolare ai parametri contenuti in quest'oggetto, il cui valore viene normalmente ricavato da una chiamata al metodo `getParameter()`.

Ecco un elenco degli attributi:

`name`

Il nome del bean, corrispondente all'attributo `id` di `jsp:useBean`, oppure a quello di uno degli oggetti impliciti.

`property`

Il nome della proprietà del bean.

`param`

Il nome del parametro dell'oggetto `request` da cui viene preso il valore da assegnare alla proprietà del bean.

`value`

Il valore della proprietà; se si vuole assegnare direttamente, questo attributo accetta request-time values sotto forma di espressioni JSP.

L'azione può essere specificata in vari modi diversi:

```
<jsp:setProperty name = "beanName" property = "propertyName" value = "explicitValue" />
```

In questo caso il valore viene assegnato direttamente specificando il valore come attributo `value`.

```
<jsp:setProperty name = "beanName" property = "propertyName" param = "parameterName" />
```

Qui il valore viene prelevato dal parametro dell'oggetto `request` specificato dall'attributo `param`; in questo modo è possibile anche assegnare interi array.

```
<jsp:setProperty name = "beanName" property = "propertyName" />
```

Il valore viene prelevato dal parametro dell'oggetto `request` con lo stesso nome della proprietà; in questo modo è possibile anche assegnare interi array.

```
<jsp:setProperty name = "beanName" property = "*" />
```

Se il valore dell'attributo `property` è un asterisco, vengono assegnate al bean tutte le proprietà per cui esiste un parametro di `request` che abbia lo stesso nome.

I valori vengono convertiti dal formato testo al tipo specifico della proprietà utilizzando il metodo `valueOf()` della classe corrispondente al tipo; ad esempio, per il tipo `int`, si userà `Integer.valueOf()`.

Se il valore assegnato al parametro è uguale a `null` o a una stringa vuota, l'assegnamento non viene effettuato e il parametro conserva il precedente valore.

L'esempio seguente mostra l'uso di `property = "*"` utilizzando un form HTML che invia dei parametri di nome uguale alle proprietà di un bean.

```
<html>
<head>
   <title>PersonalData</title>
</head>
<body>
<form action = "/mokabyte/PersonalData.jsp" method = "post">
<p>Scrivi qui sotto i tuoi dati personali:</p>
<table>
<tr>
   <td>Nome</td>
   <td><input type = "text" name = "firstName"></td>
</tr>
<tr>
   <td>Cognome</td>
   <td><input type = "text" name = "lastName"></td>
</tr>
<tr>
   <td>Telefono</td>
   <td><input type = "text" name = "telephone"></td>
</tr>
<tr>
   <td>e-mail</td>
   <td><input type = "text" name = "email"></td>
</tr>
</table>
<br><input type = "submit" value = "Procedi">
</form>
</body>
</html>
```

```
public class Person {
   String firstName;
   String lastName;
   String telephone;
   String email;
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String s) {
```

```
      firstName = s;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName(String s) {
      lastName = s;
   }
   public String getTelephone() {
      return telephone;
   }
   public void setTelephone(String s) {
      telephone = s;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String s) {
      email = s;
   }
}
```

```
<html>
<head>
   <title>PersonalData</title>
</head>
<body>
<jsp:useBean id = "bean" class = "Person" />
<jsp:setProperty name = "bean" property = "*" />
<p>Questi sono i tuoi dati personali:</p>
<table style = "font-weight: bold;">
<tr>
   <td>Nome:</td>
   <td><jsp:getProperty name = "bean" property = "firstName" /></td>
</tr>
<tr>
   <td>Cognome:</td>
   <td><jsp:getProperty name = "bean" property = "lastName" /></td>
</tr>
<tr>
   <td>Telefono:</td>
   <td><jsp:getProperty name = "bean" property = "telephone" /></td>
</tr>
<tr>
   <td>E-mail:</td>
   <td><jsp:getProperty name = "bean" property = "email" /></td>
</tr>
</table>
</body>
```

```
</html>
```

L'aver usato per gli oggetti di input HTML nomi uguali a quelli delle proprietà del bean ha consentito di assegnare tutte le proprietà con un'unica azione `jsp:setProperty`. Una volta assegnate le proprietà al bean, esse possono essere visualizzate con `jsp:getProperty`.

Il prossimo esempio mostra invece come assegnare da una pagina HTML una proprietà di tipo array.

```
<html>
<head>
   <title>Colors</title>
</head>
<body>
<form action = "/mokabyte/colors.jsp" method = "post">
<p>Scrivi qui sotto i tuoi cinque colori preferiti</p>
1.<input type = "text" name = "colors"><br>
2.<input type = "text" name = "colors"><br>
3.<input type = "text" name = "colors"><br>
4.<input type = "text" name = "colors"><br>
5.<input type = "text" name = "colors"><br>
<br><input type = "submit" value = "Procedi">
</form>
</body>
</html>
```

Nel file HTML si genera un parametro di tipo array assegnando lo stesso nome a una serie di oggetti di input.

```
import java.util.Vector;
public class Array {
   Vector vector = new Vector();

   public String[] getItems() {
      String[] items = new String[vector.size()];
      vector.toArray(items);
      return items;
   }
   public void setItems(String[] items) {
      vector.clear();
      for (int i = 0; i < items.length; i++)
         vector.addElement(items[i]);
   }

   public String getItems(int i) {
      return (String)vector.elementAt(i);
   }
   public void setItems(int i, String item) {
```

```
      vector.setElementAt(item, i);
   }
   public String getFormattedItems() {
      StringBuffer buffer = new StringBuffer();
      for (int i = 0; i < vector.size(); i++)
         buffer.append(Integer.toString(i+1) + ". "
                      + (String)vector.elementAt(i) + '\n');
      return buffer.toString();
   }
}
```

Nel bean viene definito un metodo `getFormattedItems()` che restituisce l'intero contenuto dell'array sotto forma di stringa formattata: gli elementi sono mostrati uno per riga con il valore preceduto dal numero d'ordine.

```
<html>
<head>
   <title>Colors</title>
</head>
<body>
<jsp:useBean id = "bean" class = "Array" />
<jsp:setProperty name = "bean" property = "items" param = "colors" />
Questi sono i tuoi colori preferiti:
<b>
<pre>
<jsp:getProperty name = "bean" property = "formattedItems" />
</pre>
</b>
</body>
</html>
```

Nel file JSP si usa `jsp:setProperty` per assegnare il parametro `colors` alla proprietà `items` del bean, che è di tipo `String[]`. Poiché in questo caso il nome della proprietà è diverso da quello del parametro, è necessario specificarli entrambi.

La presenza della proprietà `formattedItems` consente di visualizzare l'intero contenuto dell'array con una semplice azione `jsp:getProperty`. L'alternativa sarebbe usare uno scriptlet Java:

```
<%
String[] colors = bean.getItems();
for (int i = 0; i < colors.length; i++)
   out.println(Integer.toString(i+1) + ". " + colors[i]);
%>
```

In tal modo si hanno i soliti svantaggi del mischiare codice Java e HTML: minore leggibilità, manutenzione difficoltosa, mancanza di separazione tra programmazione e presentazione.

Si nota però che in questo caso il metodo `getFormattedItems()` sconfina nel terreno della presentazione dei dati, poiché stabilisce un formato predefinito di visualizzazione che non può essere modificato nel file JSP. Per rimediare a tale inconveniente, occorrerebbe aggiungere nel bean altre proprietà che consentano di definire il formato della stringa restituita da `getFormattedItems()`, oppure implementare dei custom tags.

L'azione `jsp:include`

Con questa azione è possibile inserire il contenuto di una pagina statica o dinamica. A differenza della *direttiva* `include`, l'inclusione avviene in fase di processing, il che consente l'inserimento di pagine dinamiche, ad esempio altre pagine JSP o pagine generate da Servlet. Con le pagine dinamiche è possibile specificare dei parametri che verranno usati per la elaborazione della risposta. Questi sono gli attributi dell'azione:

`page`

URL della pagina; per questo attributo si può usare un *request-time value*.

`flush`

Questo attributo *deve* essere specificato. Se il valore è `true` viene effettuato il flush dell'output dopo l'inclusione; nelle specifiche JSP 1.1 il valore *deve* essere `true`. Il valore `false` è presumibilmente riservato per successive versioni.

I parametri vengono specificati con un'azione `jsp:param`, come nell'esempio seguente.

```
<html>
<head>
   <title>IncludeHello</title>
</head>
<body style = "font-size: 20pt; font-weight: bold;">
<p style = "font-size: 14pt; font-weight: normal;">
Ora includo il file Hello.jsp, passandogli un parametro:
</p>
<jsp:include page = "Hello.jsp" flush = "true">
   <jsp:param name = "name" value = "a tutti"/>
</jsp:include>
</body>
</html>
```

L'azione `jsp:forward`

Con `jsp:forward` possiamo inoltrare la richiesta in fase di elaborazione a un'altra

pagina JSP o Servlet. Quest'azione determina la fine dell'esecuzione della pagina in cui compare, quindi eventuali altri contenuti saranno ignorati. Poiché l'oggetto `request` viene passato alla nuova pagina, i parametri e gli altri dati contenuti nell'oggetto saranno preservati nella nuova pagina. La risposta sarà invece totalmente a carico della nuova pagina. Infatti, se lo stream di output è bufferizzato (si veda la direttiva `page`, attributo `buffer`), il contenuto del buffer viene cancellato prima di inoltrare la richiesta; se non è bufferizzato e la pagina ha già scritto qualcosa nello stream, viene generata un'eccezione. Infatti il meccanismo di forwarding, a causa di limitazioni dovute al protocollo HTTP, non consente di inoltrare una richiesta se sono già stati inviati dati al client.

Si riporta ora un esempio di come può essere usato questo tipo di azione. Si immagini di voler fare un'indagine sulla situazione lavorativa di un certo numero di persone. La pagina seguente è il punto di partenza: un form in cui inserire dati anagrafici e condizione lavorativa.

```
<html>
<head>
   <title>Indagine sul lavoro</title>
</head>
<body>
<form action = "/mokabyte/Survey.jsp" method = "post">
<p><b>Per favore, inserisci i tuoi dati</b></p>
Nome <input type = "text" name = "firstName"><br>
Cognome <input type = "text" name = "lastName"><br>
Età <input type = "text" name = "age"><br>
<p><b>Situazione lavorativa</b></p>
<input type = "radio" name = "job" value = "lavoratore"> Lavoratore<br>
<input type = "radio" name = "job" value = "studente"> Studente<br>
<input type = "radio" name = "job" value = "disoccupato"> Disoccupato<br>
<p><input type = "submit" value = "Continua"></p>
</form>
</body>
</html>
```

I dati del form vengono mandati alla pagina `Survey.jsp`, che inoltra la richiesta ad altre pagine JSP, a seconda della condizione lavorativa indicata.

```
<html>
<head>
   <title>Indagine sul lavoro</title>
</head>
<body>
<%
   String job = request.getParameter("job");
   String nextPage = null;
   if (job.equals("lavoratore"))
```

```
      nextPage = "Worker.jsp";
   else if (job.equals("studente"))
      nextPage = "Student.jsp";
   else if (job.equals("disoccupato"))
      nextPage = "Unemployed.jsp";
%>

<jsp:forward page = "<%= nextPage %>"/>
</body>
</html>
```

La pagina successiva riceve la richiesta con i parametri precedentemente impostati, e chiede altri dati, a seconda che si tratti di un lavoratore, di uno studente o di un disoccupato. Ad esempio, per il lavoratore si avrà la seguente pagina.

```
<html>
<head>
   <title>Lavoratore</title>
</head>
<body>
<form action = "/mokabyte/SurveyResult.jsp" method = "post">
<p>
Ora inserisci qualche informazione sul tuo lavoro:<br>
</p>
<p>Tipo di attività:</p>
<p>
<input type = "radio" name
= "jobType" value = "operaio"> operaio<br>
<input type = "radio" name
= "jobType" value = "impiegato"> impiegato<br>
<input type = "radio" name
= "jobType" value  = "artigiano"> artigiano<br>
<input type = "radio" name
= "jobType" value = "professionista"> professionista<br>
<input type = "radio" name
= "jobType" value = "imprenditore"> imprenditore<br>
</p>
<p>Settore di attività:</p>
<p>
<input type = "radio" name
= "jobSector" value  = "agricoltura"> agricoltura<br>
<input type = "radio" name
= "jobSector" value = "industria"> industria<br>
<input type = "radio" name
= "jobSector" value = "artigianato"> artigianato<br>
<input type = "radio" name
= "jobSector" value = "commercio"> commercio<br>
<input type = "radio" name
= "jobSector" value = "servizi"> servizi<br>
```

```
<input type = "radio" name =
"jobSector" value = "pubblico"> pubblica amministrazione<br>
</p>
<p><input type = "submit" value = "Continua"></p>
<%@ include file = "SurveyParameters.inc" %>
</form>
</body>
</html>
```

Per i disoccupati verrà mostrata quest'altra pagina.

```
<html>
<head>
   <title>Disoccupato</title>
</head>
<body>
<form action = "/mokabyte/SurveyResult.jsp" method = "post">
<p>Ora inserisci qualche altra informazione:</p>
<p>Da quanto tempo sei disoccupato:</p>
<p>
<input type = "radio" name
= "joblessPeriod" value = "breve"> meno di 6 mesi<br>
<input type = "radio" name
= "joblessPeriod" value = "medio"> meno di 18 mesi<br>
<input type = "radio" name
= "joblessPeriod" value = "lungo"> più di 18 mesi<br>
</p>
<p>Sei iscritto alle liste di collocamento?:</p>
<p>
<input type = "radio" name = "employmentList" value = "sì"> si<br>
<input type = "radio" name = "employmentList" value = "no"> no<br>
</p>
<p><input type = "submit" value = "Continua"></p>
<%@ include file = "SurveyParameters.inc" %>
</form>
</body>
</html>
```

E per gli studenti sarà visualizzata ancora un'altra pagina, impostata come le precedenti.

Poiché in questo caso la richiesta viene inoltrata da un form HTML, non si può usare `jsp:forward` per inoltrare la stessa richiesta; con il *submit* del form verrà quindi creata una nuova richiesta. Perciò occorre usare un altro sistema per passare i parametri ricevuti dalla pagina precedente: si utilizza una serie di hidden input HTML il cui valore è rappresentato da espressioni JSP. Poiché gli hidden input sono sempre gli stessi per le tre pagine, si usa una direttiva `include` che inserisce il file `SurveyParameters.inc`, che contiene il codice HTML.

```
<input type = "hidden" name
= "firstName" value = "<%= request.getParameter("firstName") %>">
<input type = "hidden" name
= "lastName" value = "<%= request.getParameter("lastName") %>">
<input type = "hidden" name
= "age" value = "<%= request.getParameter("age") %>">
<input type = "hidden" name
= "job" value = "<%= request.getParameter("job") %>">
```

La nuova richiesta viene infine raccolta dalla pagina `SurveyResult.jsp`, che mostra i dati inseriti dall'utente.

```
<html>
<head>
   <title>Indagine sul lavoro</title>
</head>
<body>
<% String job = request.getParameter("job"); %>
<p>Questi sono i dati che hai inserito:</p>
Nome: <%= request.getParameter("firstName") %><br>
Cognome: <%= request.getParameter("lastName") %><br>
Età: <%= request.getParameter("age") %><br>
Attività: <%= job %><br>
<% if (job.equals("lavoratore")) { %>
Tipo di lavoro:  <%= request.getParameter("jobType") %><br>
Settore di attività:  <%= request.getParameter("jobSector") %><br>
<% } else if (job.equals("studente")) { %>
Tipo di scuola: <%= request.getParameter("schoolType") %>
                <%= request.getParameter("schoolOwnership") %><br>
Settore di studio: <%= request.getParameter("studySector") %><br>
<% } else if (job.equals("disoccupato")) { %>
Periodo di disoccupazione: 
<%= request.getParameter("joblessPeriod") %><br>
Iscritto alle liste di collocamento: 
<%= request.getParameter("employmentList") %><br>
<% } %>
</body>
</html>
```

Per brevità sono stati usati degli script element ma, come in altri casi illustrati precedentemente, è possibile modificare quest'esempio utilizzando bean e action element, migliorando notevolmente il design dell'applicazione, con i vantaggi già menzionati.

Le specifiche dicono che la richiesta può essere mandata con `jsp:forward` anche a una "risorsa statica". Dal momento che una pagina statica non può elaborare alcuna richiesta, l'unico risultato sarà la visualizzazione della pagina, come con un link HTML. Bisogna però fare attenzione al fatto che, se una pagina statica riceve una richiesta con il metodo `post`, questo può generare un errore `405 (method not allowed)` da parte del web server. Quindi, se la pagina JSP che visualizza la pagina statica con `jsp:forward` ha ricevuto a sua volta una richiesta da un form con il metodo `post`, si potrà ottenere un errore, poiché la richiesta sarà inoltrata con lo stesso metodo.

L'azione `jsp:param`

Si usa con `jsp:include` o `jsp:forward` per inserire nuovi parametri nella richiesta che viene inoltrata alla pagina da includere o visualizzare separatamente. Si inserisce all'interno del body dell'azione principale. Questi gli attributi:

`name`

Nome del parametro.

`value`

Valore da assegnare al parametro.

Nel caso di `jsp:include`, i nuovi parametri saranno validi solo per la pagina inclusa e non saranno utilizzabili nella pagina corrente dopo l'azione `jsp:include`. Se `jsp:param` specifica un valore per un parametro già esistente, il nuovo valore diverrà il primo elemento dell'array di valori e gli altri valori saranno conservati. Se ad esempio la richiesta contiene un parametro `A = "x"` e si assegna ad `A` un valore `"y"` con `jsp:param`, il nuovo valore di `A` sarà `"y, x"`.

L'azione `jsp:plugin`

Con `jsp:plugin` si può inserire nella pagina un'Applet o un bean, utilizzando i tag HTML `embed` o `object`, a seconda del browser che invia la richiesta. Il componente viene mandato in esecuzione per mezzo del Java Plugin che deve essere presente nel browser richiedente; se il plugin non è presente può essere effettuato automaticamente il download da un determinato URL (si vedano più avanti gli attributi).

Questo elemento può contenere all'interno del body due sottoelementi opzionali, ambedue dotati a loro volta di body:

`jsp:params`

Contiene una serie di elementi `jsp:param` che vengono passati come parametri dell'Applet o proprietà del bean.

`jsp:fallback`

All'interno si può inserire un testo arbitrario da visualizzare nel caso che il Java Plugin non possa essere fatto partire. Se il plugin entra in funzione ma non è possibile mandare in esecuzione l'Applet o il bean, sarà il plugin a visualizzare un messaggio d'errore.
Questo è l'elenco degli attributi:

`type`

Tipo di oggetto da inserire: `applet` o `bean`.

`jreversion`

Versione del Java Runtime Environment richiesto per l'esecuzione del componente. Il valore di default è `1.1`.

`nspluginurl`

URL da cui effettuare il download del Java Plugin per Netscape; il valore di default dipende dall'implementazione.

`iepluginurl`

URL da cui effettuare il download del Java Plugin per Internet Explorer; il valore di default dipende dall'implementazione.
I seguenti attributi corrispondono agli omonimi parametri HTML relativi ai tag `EMBED` o `OBJECT`. Per informazioni sull'uso e il significato, vedere le specifiche HTML dei suddetti tag:

`name`

`title`

`code`

`codebase`

`archive`

`align`

`width`

`height`

`hspace`

`vspace`

L'esempio che segue mostra come può essere visualizzata un'ipotetica Applet `Clock`, che visualizza un orologio per mezzo di componenti `Swing`, simulando un orologio analogico oppure uno digitale con display a cristalli liquidi, e accetta un parametro che specifica il tipo di orologio (`analogic` o `lcd`):

```
<jsp:plugin
   type = applet
   code = "Clock.class"
   codebase = "/jsp/applet"
   jreversion = "1.2"
   width = "150"
   height = "150""
>
   <jsp:params>
      <jsp:param name = "clockType" value = "analogic" />
   </jsp:params>
   <jsp:fallback>
      <p>Impossibile eseguire il Java Plugin</p>
   </jsp:fallback>
</jsp:plugin>
```

Poiché l'Applet usa le classi `Swing`, va specificato che la versione del runtime Java deve essere almeno la 1.2.

Directive

La direttiva `page`

La direttiva `page` contiene parametri che vengono applicati alla *translation unit* corrente, cioè alla pagina corrente e a tutti i file inclusi con la direttiva include. Non ha invece alcun effetto sui contenuti inclusi dinamicamente con `jsp:include`.

La direttiva `page` può essere usata più volte nella stessa translation unit, ma ciascun attributo deve essere specificato una sola volta; quindi ogni direttiva deve contenere attri-

buti differenti. Fa eccezione l'attributo `import`, che può essere specificato più volte, con effetto cumulativo. La posizione in cui sono inserite direttive all'interno della translation unit non influisce sul risultato, dal momento che ogni parametro è valido per l'intera unità. È comunque consigliabile, per ragioni di chiarezza, inserire la direttiva `page` all'inizio della pagina.

Gli attributi della direttiva `page` sono i seguenti:

`language`

Indica il linguaggio di script usato negli script element. Il valore di default è `java`. I linguaggi utilizzabili, oltre a Java che *deve* essere supportato, dipendono dall'implementazione, quindi variano da container a container.

`extends`

Specifica il nome completo (incluso il package) della classe di cui viene creata un'istanza per implementare in Java la pagina JSP. Il default dipende dall'implementazione. La classe deve comunque implementare l'interfaccia `JspPage`. Questo attributo si usa raramente, solo nel caso l'utente abbia implementato una propria classe per l'elaborazione delle richieste delle pagine JSP.

`import`

Una lista di packages di cui si vuole effettuare l'importazione per l'esecuzione degli script Java contenuti nella pagina. La sintassi e il risultato sono gli stessi della *import declaration* del linguaggio Java, con la differenza che si possono specificare più package insieme, separati da virgola. L'attributo `import` è l'unico che può essere usato più volte in una stessa pagina JSP. Un'altra differenza è che in una pagina JSP viene effettuata l'importazione automatica non solo del package `java.lang`, ma anche di `javax.servlet`, `javax.servlet.jsp` e `javax.servlet.http`. L'attributo `import` fa eccezione anche per lo `scope` sul quale agisce: infatti non ha effetto su tutto il contenuto della pagina, ma deve essere specificato *prima* dello script che usa le classi importate.

`session`

Il valore può essere `true` (default) o `false`. Se il valore è `true` la pagina potrà utilizzare la sessione HTTP corrente dell'applicazione; se questa non esiste, viene creata. Si ricor-

da che la sessione può essere condivisa non solo con altre pagine JSP, ma anche con normali Servlet. Se il valore è `false` l'oggetto `session` non sarà disponibile. Di conseguenza non si potrà neppure utilizzare un bean con `scope = session`.

`buffer`

Indica se l'output stream deve essere bufferizzato e, nel caso, la dimensione minima del buffer (cioè il container può anche generare un buffer di dimensioni maggiori) in kilobyte con il suffisso `kb` (ad esempio `10kb`). Il valore di default è `8kb`. Se si vuole usare uno stream non bufferizzato, si deve indicare il valore `none`. In questo caso verrà usato direttamente lo stream — restituito da `getWriter()` — dell'oggetto `response`. Altrimenti ogni operazione di scrittura sarà diretta a un oggetto locale di tipo `JspWriter`, un particolare tipo di `Writer` che supporta l'opzione di autoflush, descritta qui sotto. L'uso dell'output bufferizzato è particolarmente utile per la ridirezione (ad esempio con l'azione `jsp:forward`). Infatti può capitare che la ridirezione venga decisa a runtime dopo che dei dati sono già stati inviati in output (il caso più comune è quello delle *error pages*, descritte più avanti, che si servono appunto della ridirezione). In mancanza di bufferizzazione ogni tentativo di ridirezione dopo la scrittura di dati in output dà origine a un'eccezione.

`autoFlush`

I valori possibili sono `true` e `false`. Il default è `true`. Se il valore è `true`, quando il buffer è pieno, viene automaticamente effettuato un flush dello stream (in sostanza il contenuto del `JspWriter` viene inviato al `ServletWriter` dell'oggetto `request`). Se il valore è `false`, verrà generata un'eccezione di overflow non appena si tenterà di scrivere oltre la dimensione del buffer. È illegale impostare `autoFlush` a `false` se l'attributo buffer ha valore `none`.

`isThreadSafe`

I valori possibili sono `true` (default) e `false`. Indica se la pagina è da considerarsi thread-safe. È compito di chi scrive la pagina inserire il codice di sincronizzazione, se necessario. Se il valore è `true`, il JSP container potrà mandare più richieste concorrenti alla pagina JSP. Se il valore è `false`, il container manderà le richieste una alla volta.

`info`

Specifica una stringa di contenuto arbitrario inserita in qualità di *Servlet info* nella pagina compilata: essa è accessibile con il metodo `getServletInfo()` dell'oggetto `page (this)`.

`errorPage`

Il valore specificato deve essere un *relative URL*. Indica la pagina alla quale vengono mandate le eccezioni (si veda "Gestione degli errori").

`isErrorPage`

Il valore può essere `true` o `false` (default). Indica se la pagina corrente è una error page. Se il valore è `true`, all'interno della pagina è disponibile l'oggetto implicito `exception`.

`contentType`

Indica il MIME type e il tipo di codifica dei caratteri usata per la risposta. Il valore di default è `text/html; charset = ISO-8859-1`.

La direttiva `include`

La direttiva `include` serve per inserire il contenuto di un file all'interno della pagina JSP. Come per le altre direttive, l'esecuzione viene effettuata a *translation time*, e il contenuto inserito è statico, ossia non può essere modificato con attributi della direttiva. Il che naturalmente non significa che il contenuto incluso non possa contenere elementi dinamici JSP. La differenza rispetto all'azione `jsp:include` consiste nel fatto che quest'ultima, eseguita in fase di processing (dal Servlet compilato) dà la possibilità di includere anche pagine "virtuali" ossia non esistenti nel file system, ma corrispondenti a indirizzi virtuali interpretati del web server (ad esempio un Servlet); con `jsp:include` è inoltre possibile specificare dei parametri da inviare a pagine dinamiche.

Attributi:

`file`

L'URL del file che *deve* essere specificato come URL relativo, ossia o relativo alla locazione della pagina in cui compare la direttiva, oppure, con uno slash davanti, relativo alla root directory dell'applicazione.

Un esempio di utilizzazione di questa direttiva si trova nel precedente esempio *Survey* (file `Worker.jsp` e `Unemployed.jsp`).

La direttiva `taglib`

Questa direttiva permette di utilizzare action element non standard inclusi in una *tag library*, implementata secondo le specifiche JSP 1.1 (si veda *Tag extension*). Gli attributi sono:

`uri`

È lo *Uniform Resource Identifier* che identifica univocamente la tag library. Può essere un URL (Uniform Resource Locator) o un URN (Uniform Resource Name) come specificato dalle specifiche W3C (RTF 2396), oppure un pathname relativo.

`prefix`

È il prefisso che identifica un namespace XML, usato per evitare conflitti tra nomi di tag di diversa origine. Il prefisso è obbligatorio. I prefissi `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, `sunw` sono riservati alla Sun e non sono utilizzabili per custom tags.

Le specifiche JSP 0.92 definivano dei tag standard privi di prefisso. Alcuni sono stati sostituiti dalle standard action con prefisso `jsp`. Altri, come il tag LOOP, sono scomparsi nelle nuove specifiche. È presumibile che un tag equivalente a questo e altri tag di uso generale diverranno parte di una o più tag library standard che verranno introdotte con le future specifiche JSP, a partire dalla 1.2.

Gestione degli errori

La presenza di errori in una pagina JSP può essere ricondotta a diverse tipologie.

Errori in fase di traduzione–compilazione

Questi errori si traducono generalmente nell'invio da parte del server HTTP di un `Server Error (errore 500)`. La pagina dovrebbe riportare anche i messaggi generati dal JSP container, per errori di sintassi JSP, o dal compilatore Java per errori negli script element.

Errori in fase di processing (*runtime errors*)

Gli errori runtime si risolvono generalmente nella generazione di eccezioni Java. Queste eccezioni, come per le normali classi Java, possono essere gestite dal programmatore, oppure lasciate alla gestione di default del JSP container. Se le eccezioni vengono gestite dal programmatore, questo può scegliere di farlo direttamente nella pagina, oppure all'interno di bean utilizzati nella pagina. Ma c'è anche un'altra possibilità, che in molti casi permette una gestione più ordinata e razionale: l'uso delle *error page*.

Error page

Per utilizzare il meccanismo delle error page è sufficiente compiere le seguenti operazioni:

- Scrivere una pagina JSP per la gestione degli errori, inserendo la direttiva page con l'attributo `isErrorPage` impostato a `true`. In questa pagina sarà disponibile l'oggetto implicito exception, utilizzabile per gestire opportunamente l'eccezione.

- Inserire, in tutte le pagine JSP che devono usare la pagina precedente per la gestione delle eccezioni, una direttiva page con l'attributo `errorPage` impostato all'URL della error page.

L'esempio che segue mostra le diverse possibilità. Inizialmente compare una pagina in cui si può scegliere il modo di gestire l'eccezione:

- lasciare il compito al JSP container;

- gestire l'eccezione all'interno della stessa pagina;

- utilizzare una error page.

```
<html>
<head>
   <title>ExceptionHandling</title>
</head>
<body>
<form action = "/mokabyte/HelloQuery.jsp" method = "post">
<p>Seleziona un'opzione:</p>
<p>
<input type = "radio" name  = "formTarget" value = "Hello1.jsp" checked>
 Nessuna gestione degli errori<br>
<input type = "radio" name  = "formTarget" value = "Hello2.jsp">
 Errore gestito nella pagina<br>
<input type = "radio" name = "formTarget" value = "Hello3.jsp">
 Errore gestito con error page<br>
</p>
<input type = "submit" value = "Continua">
</form>
</body>
</html>
```

La pagina HTML inoltra una richiesta a una pagina JSP che visualizza un campo di input in cui l'utente inserisce il proprio nome. Quando si effettua un submit del form

senza aver prima inserito il nome, nella pagina a cui il form è destinato (che dipende dal valore del parametro `formTarget` impostato nella pagina iniziale) verrà generato un errore.

```
<html>
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 16pt;">
<form action
= "<%=  request.getParameter("formTarget") %>" method = "post">
Scrivi il tuo nome
<input type = "text" name = "name">
<input type = "submit" value = "Clicca qui">
</form>
</body>
</html>
```

A questo punto, ognuna delle pagine gestisce l'errore in un modo diverso. Negli esempi, per semplicità, l'eccezione viene lanciata direttamente nella pagina, ottenendo un codice piuttosto artificioso, giustificato dai fini esemplificativi. Normalmente si ha a che fare con eccezioni generate dalle classi delle librerie Java o di bean usati nella pagina, di cui spesso si rende necessaria una gestione ad hoc.

La pagina `Hello1.jsp` lancia un'eccezione senza effettuare nessun `catch`, quindi l'eccezione sarà gestita secondo il meccanismo di default implementato dal container.

```
<html>
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 20pt;">
<%
String name = request.getParameter("name");
if (name == null || name.length() == 0)
   throw new java.lang.Exception("Nome non inserito");
%>
Ciao <%= name %>!
</body>
</html>
```

La pagina `Hello2.jsp` lancia l'eccezione e la gestisce direttamente in un blocco catch, visualizzando un messaggio.

```
<html>
```

```
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 20pt;">
<%!
String getName(ServletRequest request) throws Exception {
   String name = request.getParameter("name");
   if (name == null || name.length() == 0)
      throw new java.lang.Exception("Nome non inserito");
   return name;
}
%>
<% try {
   String name = getName(request);
%>
Ciao <%= name %>!
<% } catch (java.lang.Exception e) { %>
Torna alla pagina precedente e inserisci il tuo nome!
<% } %>
</body>
</html>
```

La pagina `Hello3.jsp` lancia un'eccezione e ne affida la gestione a una error page.

```
<%@ page errorPage = "HelloError.jsp"%>
<html>
<head>
   <title>Hello</title>
</head>
<body style = "font-size: 20pt;">
<%
String name = request.getParameter("name");
if (name == null || name.length() == 0)
   throw new java.lang.Exception("Nome non inserito");
%>
Ciao <%= name %>!
</body>
</html>
```

Infine, ecco la error page che visualizza il messaggio associato all'eccezione.

```
<%@ page isErrorPage = "true" %>
<html>
<head>
   <title>HelloError</title>
</head>
<h1>Errore</h1>
```

```
<body>
<h2><%= exception.getMessage() %></h2>
<p>Tornare alla pagina precedente con il comando
        "indietro" del browser.</p>
</body>
</html>
```

Quoting convention

Le specifiche JSP definiscono una serie di *quoting convention* per poter inserire nel testo le sequenze di caratteri che normalmente identificano gli elementi JSP, e che quindi vengono interpretati dal parser. Ecco un elenco delle sequenze e delle relative versioni *quoted*:

```
<%      <\%
%>      %/>
'       \'
"       \"
```

Si nota che l'uso del *single quote* (apice singolo) e del *double quote* (doppi apici) segue la sintassi XML, per cui i valori degli attributi possono essere inseriti sia entro singoli che doppi apici. Di conseguenza raramente ci sarà la necessità di ricorrere al quoting di questi caratteri, poiché possono essere usati per fare l'uno il quoting dell'altro. Le versioni quoted dei simboli `<%` e `%>` possono essere usate da qualunque parte, anche entro le stringhe che definiscono i valori degli attributi.

Sintassi alternativa XML

Ogni elemento JSP che non segua già la sintassi XML ha una sua corrispondente forma sintattica XML. Questo fa sì che una pagina JSP possa essere definita anche come un documento XML, permettendo così, tra l'altro, l'uso di tool basati su XML per la gestione e l'authoring di pagine JSP.

Le specifiche 1.1 non richiedono che un container accetti le pagine in questa forma, ma questo requisito verrà reso obbligatorio in una successiva versione delle specifiche.

Una pagina JSP in forma XML ha la dichiarazione di tipo di documento riportata di seguito

```
<! DOCTYPE root
   PUBLIC "-//Sun Microsystems Inc.//DTD JavaServer Pages Version 1.1//EN"
   "http://java.sun.com/products/jsp/dtd/jspcore_1_0.dtd">
```

e come root element il tag `jsp:root`, che ha uno o più attributi `xmlns`, con cui si specificano il namespace standard e altri eventuali relativi a tag extension library

```
<jsp:root
   xmlns:jsp = "http://java.sun.com/products/jsp/dtd/jsp_1_0.dtd">
   ...
</jsp:root>
```

Gli altri elementi, ad eccezione delle action, già in forma XML, vanno specificati secondo le seguenti forme equivalenti XML.

```
<%@ page ... %>              <jsp:directive.page ... />
<%@ taglib ... %>            attributo xmlns in jsp:root
<%@ include ... %>           <jsp:directive.include .../>
<%! ... %>                   <jsp:declaration>...</jsp:declaration>
<% ... %>                    <jsp:scriptlet>...</jsp:scriptlet>
<%= .... %>                  <jsp:expression> ... </jsp:expression>
attr = "<%= expr %>"         attr"%= expr %"
```

Tag extension

Un'importante innovazione delle specifiche 1.1 di JSP è la definizione delle *tag extension*, ossia di un meccanismo attraverso il quale il programmatore può estendere JSP con propri custom tag (che diverranno action JSP) che svolgono particolari compiti decisi dal programmatore. L'uso di custom tag anziché di codice inserito nelle pagine ha il vantaggio di consentire l'uso di una sintassi XML e di eliminare tutti i dettagli implementativi in Java dalla pagina. Inoltre in alcuni casi i custom tag hanno una maggiore flessibilità e maneggevolezza rispetto ai bean, manipolabili solo attraverso le proprietà.

Si riprenda l'esempio `colors.jsp`. In quel caso ci si era trovati di fronte l'alternativa di inserire direttamente il codice di visualizzazione nella pagina, con uno scriptlet ("sporcando" in tal modo la pagina JSP), oppure implementare un metodo `FormattedItems()` che restituisse una stringa già nel formato di visualizzazione; con quest'ultima soluzione, però, si introduceva un indebito "congelamento" del formato di visualizzazione all'interno del codice del bean. In entrambi i casi non si otteneva una soddisfacente separazione tra elaborazione dei dati e presentazione. Una possibile strada sarebbe quella di un bean specializzato nella formattazione, che in parte si assuma i compiti normalmente svolti con HTML; ma le tag extension forniscono un meccanismo più semplice e più integrato in un linguaggio basato sui tag, come HTML.

Si veda innanzitutto come si presenta la pagina `colors.jsp` nella variante basata su tag extension.

```
<%@ taglib uri = "http://www.mokabyte.it/examples/jsp/taglib" prefix = "mb" %>
<html>
<head>
   <title>Colors</title>
```

```
</head>
<body>
<jsp:useBean id = "bean" class = "Array" />
<jsp:setProperty name = "bean" property  = "items" param = "colors" />
<p>Questi sono i tuoi colori preferiti:</p>
<mb:iterate beanName = "bean" property = "items">
   <mb:getItemNumber/>. <b><mb:getValue/></b><br>
</mb:iterate>
</body>
</html>
```

All'inizio del file la direttiva `taglib` specifica l'identificatore della libreria sotto forma di URI (Universal Resource Identificator), e il prefisso utilizzato come namespace XML. Lo URI deve essere definito nel file `web.xml`, un file di configurazione descritto in seguito in *Configurazione e deployment*, dove si trova anche la descrizione del Tag Library Descriptor, un file XML che descrive i componenti e le proprietà della tag library e che deve essere incluso nel deployment.

Per il resto, la differenza rispetto alla versione precedente è che, al posto del tag `getProperty` che prende la proprietà `formattedItems` del bean, si utilizza il tag `mb:iterate`, definito come custom tag. Questa azione compie un'iterazione su una proprietà di un bean, che deve essere un array o una `Collection`. I suoi attributi sono il nome del bean e il nome della proprietà. All'interno del body del tag si trovano altri due custom tag: `mb:getItemNumber` che restituisce il numero d'ordine dell'elemento corrente, e `mb:getValue`, che restituisce il valore dell'elemento corrente. In tal modo chi compone la pagina HTML può agire liberamente sulla presentazione della lista: può inserire gli elementi in una lista HTML o in una tabella, può visualizzare o meno il numero d'ordine, ecc. Tutto questo in modo semplice e intuitivo e senza inserire codice Java.

Va anche notato che i custom tag così definiti hanno un uso molto più generale di quello dell'esempio specifico: possono essere usati in tutti i casi in cui si vogliono compiere delle operazioni (in particolare di visualizzazione) su proprietà di un bean rappresentate da un array o una `Collection`.

Naturalmente il raggiungimento di questa semplicità di utilizzo ha un suo costo: l'implementazione di una tag extension library.

Tag handler

Le classi di supporto per l'implementazione di tag extension si trovano nel package `javax.servlet.jsp.tagext`.

Per implementare un custom tag si deve definire una classe, chiamata genericamente tag handler, che implementi una delle due interfacce `Tag` o `BodyTag`, definite nel package suddetto, il quale fornisce anche delle classi base che implementano le interfacce menzio-

nate, dalle quali è possibile derivare direttamente le classi `Tag`, chiamate `TagSupport` e `BodyTagSupport` (quest'ultima derivata da `TagSupport`). Si usa una o l'altra interfaccia/classe base a seconda della natura del tag (dotato di body, oppure empty tag) e a seconda del tipo di elaborazione richiesto dal body, come sarà chiarito meglio in seguito.

Questi sono i metodi definiti nell'interfaccia `Tag`:

```
void setPageContext(PageContext pc)
```

Questo metodo viene chiamato dal JSP container. La classe è responsabile di mantenere un reference al `Page Context` per eventuali usi interni.

```
void setParent(Tag parent)
```

Il metodo viene chiamato dal JSP container che passa un reference al `Tag` entro il quale è inserito il `Tag this`, che la classe ha la responsabilità di conservare.

```
Tag getParent()
```

Restituisce un reference al `Tag parent`.

```
void release()
```

Rilascia l'istanza della classe per la garbage collection. Anche questo metodo viene chiamato automaticamente.

```
int doStartTag()
```

Questo metodo viene chiamato dal container all'inizio dell'elaborazione. Se si tratta di un tag dotato di body; il metodo rappresenta il tag iniziale. Quando il metodo viene chiamato, il `PageContext`, il `Parent` e gli attributi sono già stati assegnati. Il metodo restituisce un valore che indica se e come compiere un processing del body. I valori possibili sono:

- `SKIP_BODY`: il body, se presente, viene ignorato. Questo è il valore da restituire se si tratta di un empty tag.
- `EVAL_BODY_INCLUDE`: il body viene valutato e il risultato della valutazione inseri-

to direttamente nello output stream. È valido soltanto per le classi che implementano l'interfaccia `Tag` e non `BodyTag`. Quindi un tag dotato di body non deve necessariamente implementare l'interfaccia `BodyTag`. Il tipico esempio di tag che usa questo valore di ritorno è un tag di inclusione condizionale. In questo caso il contenuto del body deve essere semplicemente copiato in output, senza ulteriori elaborazioni, e non è richiesta la creazione di un `BodyContent` (vedi sotto).

- `EVAL_BODY_TAG`: il body viene valutato e viene creato un `BodyContent`, ossia uno stream temporaneo il cui contenuto verrà inviato allo output stream del tag solo alla fine dell'elaborazione del body. Questo valore è valido solo per le classi che implementano `BodyTag`. È il caso di tag che compiono operazioni iterative o comunque compiono elaborazioni che rendono necessaria la creazione di un output buffer temporaneo per il body.

```
int doEndTag()
```

Viene chiamato dopo `doStartTag()` e dopo l'eventuale processing del body. Restituisce `SKIP_PAGE` o `EVAL_PAGE` a seconda che il resto della pagina debba essere elaborato dal container oppure ignorato. Ovviamente è presumibile che la restituzione di un valore `SKIP_PAGE` sia sottoposto a qualche condizione dipendente dalla precedente elaborazione a runtime.

L'interfaccia `BodyTag`, derivata da `Tag`, aggiunge i seguenti metodi:

```
void setBodyContent()
```

Chiamato prima del processing del body, solo nel caso questo abbia luogo. Assegna al tag un oggetto `BodyContent` che funge da buffer temporaneo per l'output del body.

```
void doInitBody()
```

Viene chiamato dopo `setBodyContent()`, soltanto alla prima valutazione del body (nel caso di iterazioni possono essercene più di una).

```
int doAfterBody()
```

Viene chiamato dopo ogni elaborazione del body. Se il valore restituito è

EVAL_BODY_TAG, verrà effettuata una nuova elaborazione del body, altrimenti, se il valore restituito è SKIP_BODY, l'elaborazione avrà termine e sarà chiamato il metodo doEndTag().

Le classi precedentemente citate TagSupport e BodyTagSupport sono classi concrete, ossia forniscono un'implementazione di default di tutti i metodi delle rispettive interfacce. Cosicché in una sottoclasse è sufficiente ridefinire i metodi strettamente necessari per implementare le operazioni specifiche del tag.

Nell'esempio sopra, il tag mb:iterate è un BodyTag, ed è implementato come sottoclasse di BodyTagSupport. Ecco il codice Java.

```
package mbtags;
import java.util.*;
import java.lang.reflect.*;
import java.beans.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class IterateTag extends BodyTagSupport {
   String beanName;
   String property;
   Object bean;
   Object[] array;
   int currentIndex = 0;
   public String getBeanName() {
      return beanName;
   }
   public void setBeanName(String s {
      beanName = s;
   }
   public String getProperty() {
      return property;
   }
   public void setProperty(String s) {
      property = s;
   }
   public Object getCurrentValue() {
      if (currentIndex < array.length)
         return array[currentIndex];
      else
         return "Out of bounds";
   }
   public void setCurrentValue(Object o) {
      array[currentIndex] = o;
   }
   public int getCount() {
      return array.length;
   }
   public int getCurrentIndex() {
```

```
      return currentIndex;
   }
   public boolean hasNext() {
      return currentIndex < array.length - 1;
   }
   public void increment() {
      currentIndex++;
   }
   public int doStartTag() throws JspException {
      bean = pageContext.findAttribute(getBeanName());
      array = getPropertyValue(property);
      if (!hasNext())
         return SKIP_BODY;
      return EVAL_BODY_TAG;
   }
   public int doAfterBody() throws JspException {
      if (!hasNext()) {
         try {
            getBodyContent().writeOut(getPreviousOut());
         } catch (Exception e) {
            throw new JspException(e.getMessage());
         }
         return SKIP_BODY;
      }
      increment();
      return EVAL_BODY_TAG;
   }
   private Object[] getPropertyValue(String propertyName) throws JspException {
      Method method = null;
      Class type = null;
      Object dummyArgs[] = new Object[0];
      Object value = null;
      try {
         BeanInfo info = Introspector.getBeanInfo(bean.getClass());
         if (info != null) {
            PropertyDescriptor pds[] = info.getPropertyDescriptors();
            PropertyDescriptor pd = null;
            for (int i = 0 ; i < pds.length ; i++) {
               pd = pds[i];
               if (pd.getName().equals(propertyName)) {
                  type = pd.getPropertyType();
                  value = pd.getReadMethod().invoke(bean, dummyArgs);
                  if (type.isArray())
                     return (Object[])value;
                  else if (type.isInstance(Collection.class))
                     return ((Collection)value).toArray();
                  else
                     throw new JspException("Not an indexed property");
               }
```

```
            }
         }
      } catch (JspException e) {
         throw e;
      } catch (Exception e) {
         throw new JspException(e.toString());
      }
      return null;
   }
}
```

La classe contiene innanzi tutto una serie di metodi `get` e `set` per gli attributi del tag (`beanName()` e `property`). Seguono alcuni metodi che restituiscono altre proprietà non esposte come attributi del tag (`value`, `count`, `currentIndex`) e due metodi usati per l'iterazione (`hasNext()` e `increment()`). Come si vedrà fra poco, alcuni di questi metodi sono utilizzati da altre classi che implementano tag che agiscono all'interno del body, in collaborazione con questa.

Dei metodi dell'interfaccia `BodyTag`, la classe implementa soltanto `doStartTag()` e `doAfterBody()`, mentre per gli altri sono utilizzate le implementazioni di default della classe `BodyTagSupport`.

Il metodo `doStartTag` recupera un reference al bean indicato dall'attributo `beanName`, che deve essere stato precedentemente creato con `jsp:useBean`, quindi chiama il metodo `getPropertyValue()` per ottenere il valore della proprietà indicata dall'attributo `property`. Questo metodo privato usa la classe `Introspector` per ottenere un array dei `PropertyDescriptor` del bean, cerca il metodo con il nome dato e, se lo trova, chiama il rispettivo metodo `get` per ottenerne il valore. Infine controlla il tipo della proprietà: se si tratta di un array lo restituisce direttamente, se si tratta di una Collection la trasforma prima in array, se il tipo è ancora diverso lancia una eccezione, poiché non lo riconosce come tipo di una proprietà indicizzabile. Tornando al metodo `doStartTag()`, una volta ottenuto il valore della proprietà, se l'array non è vuoto, restituisce `EVAL_BODY_TAG`, in modo che si proceda alla valutazione del body, altrimenti restituisce `SKIP_BODY`, facendo sì che il body venga ignorato.

Il metodo `doAfterBody()` è quello che gestisce l'iterazione: finché trova nuovi elementi nell'array, incrementa l'indice corrente; alla fine, quando non ci sono più elementi, scrive il contenuto del `BodyContent` (restituito dal metodo `getBodyContent()`) sullo stream di output (restituito dal metodo `getPreviousOut()`) utilizzando il metodo `BodyContent.writeOut()`.

Cooperazione tra azioni JSP

Come si vede dal codice, la classe `IteratorTag` non si occupa affatto di quello che accade all'interno del body: si limita a avviare il processing del body per ogni elemento

contenuto nell'array. Perché la classe sia realmente di utilità pratica è necessario definire altre classi che agiscano in collaborazione con essa, ossia classi che rappresentano altri tag che vengono usati all'interno del body. Nell'esempio sono utilizzati altri due tag all'interno di `mb:iterator`, ovvero `mb:getValue` e `mb:getItemNumber`, che restituiscono rispettivamente il valore dell'elemento corrente e il numero d'ordine dello stesso elemento. Poiché la proprietà a cui ci si riferisce è determinata da un attributo del tag `mb:iterator`, e l'elemento corrente dell'array è determinato sempre dal processo di iterazione, i due tag non hanno bisogno di attributi, ma si appoggiano interamente al tag `parent` per ottenere i rispettivi valori.

```
package mbtags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class GetValueTag extends TagSupport {
   public int doStartTag() throws JspException {
      IterateTag parent = (IterateTag)getParent();
      if (parent == null)
         throw new JspException("null parent of getValue tag");
      try {
         Object value = parent.getValue();
         if (value == null)
            throw new JspException("null value in getValue tag");
         pageContext.getOut().print(value);
      } catch (Exception e) {
         throw new JspException(e.toString());
      }
      return SKIP_BODY;
   }
}
```

Si tratta naturalmente di empty tag, privo di body, e quindi derivato da `TagSupport`. L'unico metodo che viene ridefinito è `doStartTag()`. Il metodo assume che il parent sia un `IterateTag`, quindi fa un cast del reference restituito da `getParent()` a `IterateTag`. Dopodiché non fa altro che prendere il valore dal metodo `getValue()` e scriverlo nello stream di output, restituito da `PageContext.getOut()`. Questo metodo restituisce lo stream corrente, che nel caso specifico corrisponde al `BodyContent` creato all'inizio della elaborazione del body di `mb:iterator`.

```
package mbtags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class GetItemNumberTag extends TagSupport {
   public int doStartTag() throws JspException {
      IterateTag parent = (IterateTag)getParent();
```

```
      if (parent == null)
         throw new JspException("null parent of getValue tag");
      try {
         pageContext.getOut().print(parent.getCurrentIndex() + 1);
      } catch (Exception e) {
         throw new JspException(e.toString());
      }
      return SKIP_BODY;
   }
}
```

Questa classe è molto simile alla precedente, e manda in output il valore dell'indice corrente aumentato di uno, corrispondente al numero d'ordine dell'elemento corrente. In tutti e due i casi viene restituito `SKIP_BODY` poiché non c'è nessun body da elaborare.

In questo caso il mezzo usato per condividere dati tra diversi tag in cooperazione tra loro è stato il riferimento diretto al tag `parent`, attraverso il metodo `getParent()`. Ma in certi casi la collaborazione può essere realizzata tra tag che non hanno questo rapporto di parentela. È il caso dei tag standard `jsp:useBean`, `jsp:setProperty`, `jsp:getProperty`: il primo crea o rende disponibile un reference a un certo oggetto, gli altri due manipolano le proprietà dell'oggetto. In questo caso la condivisione avviene attraverso l'oggetto implicito `pageContext`: gli oggetti vengono inseriti come attributi di `pageContext`, e ritrovati come tali. Questo è il sistema utilizzato nel metodo `doStartTag()` della classe `IterateTag`: il reference a un oggetto precedentemente creato da `jsp:useBean` viene recuperato con `PageContext.findAttribute()`; questo metodo rintraccia un oggetto per nome con una ricerca in tutti gli scope, cominciando dal più interno, quello della pagina; invece con il metodo `getAttribute()`, la ricerca è ristretta al solo scope specificato (se nessuno scope è specificato, assume quello della pagina).

Finora si sono visti mezzi per condividere oggetti tra diversi elementi sotto forma di attributi di tag. Di seguito si descrive un metodo per creare oggetti impliciti, simili a quelli standard, che possono essere definiti da azioni, ma usati negli script.

Definizione di scripting variables

L'uso di tag extension come quelle descritte, come si può vedere dal sorgente della pagina JSP, rende il codice molto pulito, perché privo di elementi "estranei" in codice script e allo stesso tempo piuttosto conciso. Tuttavia in alcuni casi potrebbe sorgere l'esigenza di usare elementi di script, anche soltanto espressioni JSP, per compiere operazioni che risulterebbero complicate da gestire con i soli tag. In casi del genere risulterebbe utile avere a disposizione, all'interno del body, una variabile utilizzabile in uno script, come nel seguente esempio.

```
<mb:iterate beanName = "bean" property = "items">
   <mb:getItemNumber/>.  <%= currentValue.toUpperCase() %><br>
</mb:iterate>
```

In questo caso `currentValue` è una variabile che rappresenta un oggetto implicito valido soltanto all'interno del body, e in quel contesto utilizzabile in scripting elements. Per far sì che il container definisca una variabile come questa, e per assegnarle gli opportuni valori è necessario eseguire le procedure riportate di seguito.

Definire una sottoclasse della classe di libreria `TagExtraInfo` e ridefinire il metodo `getVariableInfo()` in modo che restituisca un oggetto `VariableInfo` che contenga le opportune informazioni sulla variabile.

In uno o più dei metodi `doStartTag()`, `doEndTag()`, `doInitBody()` o `doAfterBody()`, a seconda dello scope dell'oggetto `VariableInfo` (vedi sotto) e della natura particolare della variabile, inizializzare e/o valorizzare opportunamente l'attributo associato alla variabile (si veda più avanti per maggiori dettagli).

Inserire un riferimento alla sottoclasse di `TagExtraInfo` implementata nel Tag Library Descriptor (TLD), un file di configurazione che sarà descritto più avanti. Questa informazione è necessaria perché il container deve sapere in fase di traduzione quali variabili definire nel codice Java che dovrà generare.

Qui di seguito si mostra il codice Java che definisce la classe e quello che inizializza la variabile, mentre le informazioni per il TLD saranno descritte più avanti:

```
package mbtags;
import javax.servlet.jsp.tagext.*;
public class IterateTagExtraInfo extends TagExtraInfo {
   public VariableInfo[] getVariableInfo(TagData data) {
      return new VariableInfo[] {
         new VariableInfo( "currentValue",        // nome variabile
                           "String",              // tipo variabile
                           true,                  // indica nuova variabile
                           VariableInfo.NESTED)   // scope della variabile
      };
   }
}
```

In questo caso si ha una sola variabile, quindi si crea un array di oggetti `VariableInfo` contenente un unico elemento. In caso di più variabili, si inserisce un oggetto `VariableInfo` per ciascuna variabile. Il costruttore di `VariableInfo` prende una serie di argomenti che indicano il nome e il tipo della variabile, un booleano che indica se la variabile rappresenta un nuovo oggetto e quindi deve essere dichiarata (nel caso che il linguaggio di script lo richieda), e infine una costante che indica lo scope.

Quest'ultimo argomento può assumere i seguenti valori:

- `NESTED`: la variabile è visibile solo all'interno del body del tag a cui si riferisce;

- `AT_BEGIN`: la variabile è visibile subito dopo lo start tag, sia all'interno del body, sia nel resto della pagina;

- `AT_END`: la variabile è visibile dopo la fine del body, quindi solo all'esterno del tag, nel resto della pagina.

La variabile viene inizializzata e aggiornata ad ogni ciclo nei metodi `doStartTag()` e `doAfterBody()`, che risulteranno modificati come segue.

```
public int doStartTag() throws JspException {
   bean = pageContext.findAttribute(getBeanName());
   array = getPropertyValue(property);
   if (!hasNext())
      return SKIP_BODY;
   pageContext.setAttribute("currentValue", getCurrentValue());
   return EVAL_BODY_TAG;
}
public int doAfterBody() throws JspException {
   if (!hasNext()) {
      try {
         getBodyContent().writeOut(getPreviousOut());
      } catch (Exception e) {
         throw new JspException(e.getMessage());
      }
      return SKIP_BODY;
   }
   increment();
   pageContext.setAttribute("currentValue", getCurrentValue());
   return EVAL_BODY_TAG;
}
```

Il container usa la classe `TagExtraInfo` in fase di compilazione per ottenere le informazioni sulle variabili, e usa il nome assegnato all'oggetto `VariableInfo` sia per il nome della variabile Java che verrà creata, sia come nome dell'attributo del `PageContext` a cui la variabile verrà associata (in pratica, nel codice generato, alla variabile verrà assegnato il valore restituito da `PageContext.getAttribute()`). Ai metodi del tag handler spetta il compito di assegnare i valori all'attributo del `PageContext`, e al codice del Servlet, generato dal container, di assegnare il valore dell'attributo alla variabile. In ambedue i casi, l'assegnamento andrà fatto al momento opportuno, a seconda dello scope specificato.

Nell'esempio sopra, l'attributo viene dapprima inizializzato nel metodo `doStartTag()`, quindi aggiornato ad ogni iterazione nel metodo `doAfterBody()`.

Con questo sistema si possono anche definire oggetti impliciti analoghi a quelli standard, disponibili nell'intera pagina a tutti gli script. L'approccio consigliato dalle specifiche (non si tratta di una regola, ma di una raccomandazione) è di definire un tag chiamato `defineObjects` e metterlo subito dopo la direttiva `taglib`. Si noti che le variabili così create, anche se lo scope della variabile in sé è sempre limitato alla pagina, possono contenere reference a oggetti di scope più ampio, come `request` o `session`. È sufficiente che il tag handler assegni all'attributo un oggetto con lo scope desiderato, ad esempio

```
pageContext.setAttribute("attrName",
                         pageContext.getAttribute("attrName",
                                                  PageContext.SESSION_SCOPE));
```

mentre non funzionerebbe se si specificasse lo scope direttamente in `setAttribute()`, dal momento che il codice del Servlet utilizza comunque un attributo di scope `PAGE_SCOPE`.

Validazione in fase di traduzione

Un'altra funzione della classe `TagExtraInfo` è quella di dare la possibilità di definire metodi di validazione dei tag, usati dal container in fase di traduzione (mentre la validazione a request-time può essere normalmente incorporata nel codice dei bean o negli script). In parte la validazione a translation-time viene effettuata con le informazioni contenute dei file di configurazione (si veda sotto *Configurazione e deployment*) e inserite dal container in oggetti `TagLibraryInfo` e `TagInfo` (per informazioni dettagliate su questi oggetti, si veda la documentazione `javadoc` del package `javax.servlet.jsp.tagext`); ma la classe `TagExtraInfo` permette di definire regole particolari non previste dalle informazioni di configurazione.

Per questo scopo si può utilizzare il metodo `TagExtraInfo.validate()`, che riceve come argomento un oggetto di tipo `TagData`, che contiene i valori degli attributi. Naturalmente la validazione può essere fatta solo se il valore non è un'espressione che sarà valutata a request-time, nel qual caso il valore dell'attributo restituito dal metodo `getAttribute()` dell'oggetto `TagData` sarà `TagData.REQUEST_TIME_VALUE`.

Configurazione e deployment

Il deployment delle Java Server Pages segue le regole stabilite per le Web Application dalle specifiche dei Servlet 2.2 di Sun. Tali specifiche comprendono i DTD (Document Type Definition) dei Deployment Descriptor XML, che contengono informazioni sui componenti, necessarie al container per eseguire correttamente l'applicazione. Le specifiche forniscono inoltre delle raccomandazioni (che quindi non sono da considerarsi obbligatorie per una data implementazione di un Servlet/JSP Container) per la struttura delle

directory di una Web Application, e per il packaging in un file WAR, compresso, realizzabile con `jar`, ma di estensione `.war`.

Si rimanda al capitolo sui Servlet per maggiori dettagli e si riportano qui soltanto alcune informazioni specifiche relative a JSP.

Il file `web.xml`

Nel file `web.xml`, che descrive i componenti dell'applicazione, non c'è nessuna particolare informazione da includere per le pagine JSP, a meno che non utilizzino tag extension library o non siano incluse sotto forma di Servlet precompilati. In quest'ultimo caso, dopo aver compilato la pagina utilizzando i tools specifici del container, si devono includere le informazioni come per un normale Servlet, più un mapping sulla pagina JSP, come nel seguente esempio.

```
<!DOCTYPE webapp
SYSTEM "http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<webapp>
   <servlet>
      <servlet-name> HelloWorld </servlet-name>
      <servlet-class> HelloWorld.class </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name> HelloWorld </servlet-name>
      <url-pattern> /HelloWorld.jsp </url-pattern>
   </servlet-mapping>
</webapp>
```

Questo file `web.xml` si riferisce a un'applicazione web che comprende la pagina JSP `HelloWorld.jsp` sotto forma di Servlet. Si tratta di un normale Servlet con un mapping sull'URL `/HelloWorld.jsp`, relativo alla root del container.

L'esempio seguente invece è il file `web.xml` da usare in un'applicazione che usi una tag extension library:

```
<!DOCTYPE webapp
SYSTEM "http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<webapp>
   <taglib>
      <taglib-uri>
         http://www.mokabyte.it/examples/jsp/taglib
      </taglib-uri>
      <taglib-location>
         /WEB-INF/taglib/mb.tld
      </taglib-location>
   </taglib>
```

```
</webapp>
```

L'esempio si riferisce alla tag extension library descritta sopra in *Tag extension*, e usata nell'esempio `colors2.jsp`. Il tag `taglib-uri` definisce l'identificatore della tag-library, mentre il tag `taglib-location` specifica la localizzazione del Tag Library Descriptor, un altro file XML che contiene le informazioni relative alla libreria.

Il Tag Library Descriptor

Il Tag Library Descriptor è un file di configurazione in formato XML, che fornisce al container le informazioni necessarie per poter utilizzare correttamente la libreria.

Il container, con le informazioni contenute nel file XML genera un oggetto `TagLibraryInfo` e una serie di oggetti `TagInfo`. Queste due classi fanno parte del package `javax.servlet.jsp.tagext`.

Il DTD del Tag Library Descriptor definisce gli elementi descritti qui di seguito:

- `taglib`: definisce il tipo di documento, con attributi `id` (opzionale) e `xmlns` (fisso, il namespace `"http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd")`.

All'interno di `taglib` compaiono i seguenti sottoelementi:

- `tlibversion` (obbligatorio): indica il numero di versione della libreria.
- `jspversion`: indica la versione di JSP che deve essere supportata dal container. Il valore di default è 1.1.
- `shortname` (obbligatorio): un nome breve, di pochi caratteri, utilizzabile da un tool di authoring come prefisso.
- `uri`: un nome che identifica in modo univoco la specifica versione della libreria. Non è obbligatorio, in quanto lo URI viene comunque definito nel file `web.xml` per ogni applicazione.
- `info`: un testo arbitrario contenente informazioni sulla libreria.
- `tag` (obbligatorio): descrive un tag della libreria. Deve esserne presente uno per ogni tag.

I seguenti elementi sono sottoelementi dell'elemento `tag`, che descrivono le proprietà del tag:

- `name` (obbligatorio): nome del tag.
- `tagclass` (obbligatorio): classe Java usata per l'implementazione del tag handler. Deve implementare l'interfaccia `javax.sevlet.jsp.tagext.Tag`.
- `teiclass`: indica il nome della sottoclasse di `TagExtraInfo` eventualmente utilizzata per la definizione di scripting variables o per la validazione. È obbligatorio se si vuole utilizzare la classe.
- `bodycontent`: indica il tipo di contenuto del body. Può assumere uno dei seguenti valori:

  `JSP` (default): il contenuto è elaborato secondo le regole delle pagine JSP.
  `empty`: il body deve essere vuoto.
  `tagdependent`: il body è elaborato dal tag handler; in genere contiene codice in un linguaggio non interpretato dal container (ad esempio SQL).

- `info`: un testo arbitrario contenente una descrizione del tag.
- `attribute` (obbligatorio): definisce un attributo del tag. Se ne deve includere uno per ogni attributo.

I seguenti elementi sono sottoelementi dell'elemento `attribute`, che descrivono le proprietà dell'attributo:

- `name` (obbligatorio): il nome dell'attributo.
- `required`: indica se l'attributo è obbligatorio. Il valore di default è false.
- `rtexprvalue`: indica se l'attributo accetta request-time values, ossia una espressione JSP come valore. Il valore di default è false.

Si riporta qui sotto il file `mb.tld`, relativo alla libreria descritta nell'esempio.

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "- //Sun Microsystems, Inc.
  //DTD JSP Tag Library 1.1
  //EN"
   "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>mb</shortname>
```

```
   <uri>http://www.mokabyte.it/examples/jsp/taglib</uri>
   <info>
      Esempio di tag library
   </info>
   <tag>
      <name>iterate</name>
      <tagclass>mbtags.IterateTag</tagclass>
      <teiclass>mbtags.IterateTagExtraInfo</teiclass>
      <bodycontent>JSP</bodycontent>
      <info>
         Iteratore su una proprietà di un bean di tipo array o Collection
      </info>
      <attribute>
         <name>beanName</name>
         <required>true</required>
      </attribute>
      <attribute>
         <name>property</name>
         <required>true</required>
         <rtexprvalue>true</rtexprvalue>
      </attribute>
   </tag>
   <tag>
      <name>getValue</name>
      <tagclass>mbtags.GetValueTag</tagclass>
      <info>
         Restituisce il valore della proprietà specificata in un tag iterator.
         Deve essere usato all'interno del body del tag iterator.
      </info>
   </tag>
   <tag>
      <name>getItemNumber</name>
      <tagclass>mbtags.GetItemNumberTag</tagclass>
      <info>
         Restituisce il numero d'ordine della proprietà
         specificata in un tag iterator.
         Deve essere usato all'interno del body del tag iterator.
      </info>
   </tag>
</taglib>
```

Java e CORBA

DI GIANLUCA MORELLO

Introduzione

Nell'era di Internet e delle grandi Intranet aziendali, il modello computazionale dominante è chiaramente quello distribuito. Un tipico ambiente distribuito vede la presenza di mainframe, server Unix e macchine Windows e pone quindi seri problemi di interoperabilità tra piattaforme differenti, sistemi operativi differenti e linguaggi differenti.

Lo scopo dichiarato delle specifiche CORBA è proprio quello di definire un'infrastruttura standard per la comunicazione tra e con oggetti remoti ovunque distribuiti, indipendentemente dal linguaggio usato per implementarli e dalla piattaforma di esecuzione. È bene quindi notare che, a differenza di altre tecnologie distribuite quali RMI, Servlet o EJB, non si sta parlando di una tecnologia legata a una specifica piattaforma, ma di uno standard indipendente dal linguaggio adottato che consente a oggetti Java di comunicare con "oggetti" sviluppati in COBOL, C++ o altri linguaggi ancora.

L'acronimo CORBA sta per *Common Object Request Broker Architecture* e non rappresenta uno specifico prodotto, bensì un'insieme di specifiche volte a definire un'architettura completa e standardizzata di cui esistono varie implementazioni. Le specifiche sono prodotte da OMG, un consorzio che comprende più di 800 aziende e include i più illustri marchi dell'industria informatica.

L'elemento fondamentale dell'intera architettura è il canale di comunicazione degli oggetti nell'ambiente distribuito, l'*Object Request Broker* (ORB).

Le specifiche CORBA 1.1 furono pubblicate da OMG nell'autunno del 1991 e definivano un'API e un linguaggio descrittivo per la definizione delle interfacce degli oggetti CORBA detto *Interface Definition Language* (IDL). Soltanto nel dicembre del 1994, con CORBA 2.0, vennero definiti i dettagli sulla comunicazione tra differenti implementazioni di ORB con l'introduzione dei protocolli GIOP e IIOP.

Sebbene CORBA possa essere utilizzato con la maggior parte dei linguaggi di programmazione, Java risulta il linguaggio privilegiato per implementare le sue specifiche in un ambiente eterogeneo in quanto permette agli oggetti CORBA di essere eseguiti indifferentemente su mainframe, network computer o telefono cellulare.

Nello sviluppo di applicazioni distribuite, Java e CORBA si completano a vicenda: CORBA affronta e risolve il problema della trasparenza della rete, Java quello della trasparenza dell'implementazione rispetto alla piattaforma di esecuzione.

Object Management Group

L'OMG è un consorzio no-profit interamente dedicato alla produzione di specifiche e di standard; vede la partecipazione sia dei grandi colossi dell'informatica, sia di compagnie medie e piccole.

L'attività di OMG cominciò nel 1989 con soli otto membri tra cui Sun Microsystems, Philips, Hewlett-Packard e 3Com. Le specifiche più conosciute prodotte dal consorzio sono sicuramente UML e CORBA che comunque nella logica OMG, rappresentano strumenti strettamente cooperanti nella realizzazione di applicazioni Enterprise OO.

Sin da principio il suo scopo è stato quello di produrre e mantenere una suite di specifiche di supporto per lo sviluppo di software distribuito in ambienti distribuiti, coprendo l'intero ciclo di vita di un progetto: analisi, design, sviluppo, runtime e manutenzione.

Per ridurre complessità e costi di realizzazione, il consorzio ha introdotto un framework per la realizzazione di applicazioni distribuite. Questo framework prende il nome di *Object Management Architecture* (OMA) ed è il centro di tutte le attività del consorzio; all'interno di OMA convivono tutte le tecnologie OMG.

Nell'ottica OMG la definizione di un framework prescinde dall'implementazione e si "limita" alla definizione dettagliata delle interfacce di tutti i componenti individuati in OMA. I componenti di OMA sono riportati di seguito.

Object Request Broker (ORB)

È l'elemento fondamentale dell'architettura. È il canale che fornisce l'infrastruttura che permette agli oggetti di comunicare indipendentemente dal linguaggio e dalla piattaforma adottata. La comunicazione tra tutti i componenti OMA è sempre mediata e gestita dall'ORB.

Object Services

Standardizzano la gestione e la manutenzione del ciclo di vita degli oggetti. Forniscono le interfacce base per la creazione e l'accesso agli oggetti. Sono indipendenti dal singolo dominio applicativo e possono essere usati da più applicazioni distribuite.

Common Facilities

Comunemente conosciute come CORBAFacilities. Forniscono due tipologie di servizi, orizzontali e verticali. Quelli orizzontali sono funzionalità applicative comuni: gestione stampe, gestione documenti, database e posta elettronica. Quelli verticali sono invece destinati a una precisa tipologia di applicazioni.

Domain Interfaces

Possono combinare common facilities e object services. Forniscono funzionalità altamente specializzate per ristretti domini applicativi.

Application Objects

È l'insieme di tutti gli altri oggetti sviluppati per una specifica applicazione. Non è un'area di standardizzazione OMG.

Di questi componenti OMG fornisce una definizione formale sia delle interfacce (mediante IDL), sia della semantica. La definizione mediante interfacce lascia ampio spazio al mercato di componenti software di agire sotto le specifiche con differenti implementazioni e consente la massima interoperabilità tra componenti diversi di case differenti.

I servizi CORBA

I CORBAServices sono una collezione di servizi system-level descritti dettagliatamente con un'interfaccia IDL; sono destinati a completare ed estendere le funzionalità fornite dall'ORB. Forniscono un supporto che va a coprire lo spettro completo delle esigenze di una qualunque applicazione distribuita. Alcuni dei servizi standardizzati da OMG (ad esempio il Naming Service) sono diventati fondamentali nella programmazione CORBA e sono presenti in tutte le implementazioni. Altri servizi appaiono invece meno interessanti nella pratica comune, assumono un significato solo dinanzi a esigenze particolari e non sono presenti nella maggior parte delle implementazioni sul mercato.

OMG ha pubblicato le specifiche di ben 15 servizi, qui riportati.

Collection Service

Fornisce meccanismi di creazione e utilizzo per le più comuni tipologie di collection.

Concurrency Control Service

Definisce un lock manager che fornisce meccanismi di gestione dei problemi di concorrenza nell'accesso a oggetti agganciati all'ORB.

Event Service

Fornisce un event channel che consente ai componenti interessati a uno specifico evento di ricevere una notifica, pur non conoscendo nulla del componente generatore.

Externalization Service

Definisce meccanismi di streaming per il trattamento dei dati da e verso i componenti.

Licensing Service

Fornisce meccanismi di controllo e verifica di utilizzo di un componente. È pensato per l'implementazione di politiche "pago per quel che uso".

Lyfe Cycle Service

Definisce le operazioni necessarie a gestire il ciclo di vita di un componente sull'ORB (creare, copiare e rimuovere).

Naming Service

Consente a un componente di localizzare risorse (componenti o altro) mediante nome. Permette di interrogare sistemi di directory e naming già esistenti (NIS, NDS, X.500, DCE, LDAP). È il servizio più utilizzato.

Persistence Service

Fornisce mediante un'unica interfaccia le funzionalità necessarie alla memorizzazione di un componente su più tipologie di server (ODBMS, RDBMS e file system).

Properties Service

Permette la definizione di proprietà legate allo stato di un componente.

Query Service

Fornisce meccanismi di interrogazione basati su *Object Query Language*, estensione di SQL.

Relationship Service

Consente definizione e verifica dinamiche di varie associazioni e relazioni tra componenti.

Security Service

Framework per la definizione e la gestione della sicurezza in ambiente distribuito. Copre ogni possibile aspetto: autenticazione, definizione di credenziali, gestione per delega, definizione di access control list e non-repudiation.

Time Service

Definisce un'interfaccia di sincronizzazione tra componenti in ambiente distribuito.

Trader Service

Fornisce un meccanismo modello Yellow Pages per i componenti.

Transaction Service

Fornisce un meccanismo di two-phase commit sugli oggetti agganciati all'ORB che supportano il rollback; definisce transazioni flat o innestate.

Le basi CORBA

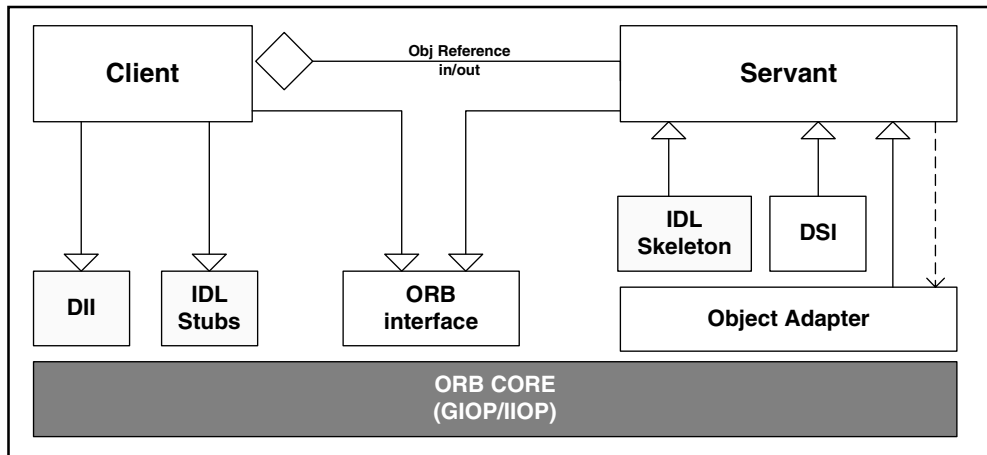
CORBA e OMA in genere si fondano su alcuni principi di design:

- separazione tra interfaccia e implementazione: un client è legato all'interfaccia di un oggetto CORBA, non alla sua implementazione;
- Location Transparency e Access Transparency: l'utilizzo di un qualunque oggetto CORBA non presuppone alcuna conoscenza sulla sua effettiva localizzazione;
- Typed Interfaces: ogni riferimento a un oggetto CORBA ha un tipo definito dalla sua interfaccia.

In CORBA è particolarmente significativo il concetto di trasparenza, inteso sia come location transparency, sia come trasparenza del linguaggio di programmazione adottato. In pratica è trasparente al client la collocazione dell'implementazione di un oggetto, locale o remota. La location transparency è garantita dalla mediazione dell'ORB. Un riferimento a un oggetto remoto va inteso come un identificativo unico di una sua implementazione sulla rete.

Architettura CORBA

L'architettura CORBA ruota intorno al concetto di Objects Request Broker. L'ORB è il

Figura 16.1 – *Architettura CORBA*

servizio che gestisce la comunicazione in uno scenario distribuito agendo da intermediario tra gli oggetti remoti: individua l'oggetto sulla rete, comunica la richiesta all'oggetto, attende il risultato e lo comunica indietro al client.

L'ORB opera in modo tale da nascondere al client tutti i dettagli sulla localizzazione degli oggetti sulla rete e sul loro linguaggio d'implementazione; è quindi l'ORB a individuare l'oggetto sulla rete e a effettuare le opportune traslazioni nel linguaggio d'implementazione. Queste traslazioni sono possibili solo per quei linguaggi per i quali è stato definito un mapping con IDL (questa definizione è stata operata per i linguaggi più comuni).

In fig. 16.1 si può osservare l'architettura CORBA nel suo complesso:

- Object è l'entità composta da identity, interface e implementation (servant nel gergo CORBA).

- Servant è l'implementazione dell'oggetto remoto. Implementa i metodi specificati dall'interfaccia in un linguaggio di programmazione.

- Client è l'entità che invoca i metodi (operation nel gergo CORBA) del servant. L'infrastruttura dell'ORB opera in modo tale da rendergli trasparenti i dettagli della comunicazione remota.

- ORB è l'entità logica che fornisce i meccanismi per inviare le richieste da un client all'oggetto remoto. Grazie al suo operato, che nasconde completamente i dettagli di comunicazione, le chiamate del client sono assimilabili a semplici invocazioni locali.

- ORB Interface: essendo un'entità logica, l'ORB può essere implementato in molti modi. Le specifiche CORBA definiscono l'ORB mediante un'interfaccia astratta, nascondendo completamente alle applicazioni i dettagli d'implementazione.

- IDL stub e IDL skeleton: lo stub opera da collante tra client e ORB; lo skeleton ha la stessa funzione per il server. Stub e skeleton sono generati nel linguaggio adottato da un compilatore apposito che opera partendo da una definizione IDL.

- Dynamic Invocation Interface (DII) è l'interfaccia che consente a un client di inviare dinamicamente una request a un oggetto remoto, senza conoscerne la definizione dell'interfaccia e senza avere un legame con lo stub. Consente inoltre a un client di effettuare due tipi di chiamate asincrone: deferred synchronous (separa le operazioni di send e di receive) e oneway (solo send).

- Dynamic Skeleton Interface (DSI) è l'analogo lato server del DII. Consente a un ORB di recapitare una request a un oggetto che non ha uno skeleton statico, ossia non è stato definito precisamente il tipo a tempo di compilazione. Il suo utilizzo è totalmente trasparente a un client.

- Object Adapter assiste l'ORB nel recapitare le request a un oggetto e nelle operazioni di attivazione/disattivazione degli oggetti. Il suo compito principale è quello di legare l'implementazione di un oggetto all'ORB.

Invocazione CORBA

Utilizzando l'ORB, un client può inviare una Request in modo trasparente a un oggetto CORBA che risieda sulla stessa macchina od ovunque sulla rete. Per raggiungere questo livello di astrazione, ogni oggetto remoto è dotato di uno stub e di uno skeleton; questi due elementi agiscono rispettivamente da collante tra client e ORB e tra ORB e oggetto CORBA.

In maniera similare a quanto accade in RMI, lo stub effettua il marshalling dei dati, traslando i data types dal linguaggio di programmazione client-side a un generico formato CORBA; quest'ultimo è convogliato via rete dal messaggio di Request.

Il client invoca i metodi non sull'oggetto remoto, bensì sul suo stub locale; l'effettiva invocazione remota viene operata dallo stub. Come si vedrà più dettagliatamente in seguito, il meccanismo dello stub è una precisa implementazione del pattern Proxy.

In maniera speculare a quanto effettuato dallo stub, l'unmarshalling dei dati è eseguito sul server dallo skeleton; in questo caso il formato della Request viene traslato nel linguaggio di programmazione server-side.

Come si è detto in precedenza, Stub e skeleton sono generati automaticamente da un compilatore a partire dalla definizione IDL dell'oggetto CORBA.

Interfaccia e funzionalità di un ORB

L'interfaccia di un ORB è definita dalle specifiche CORBA. La maggior parte degli ORB forniscono alcune operazioni addizionali, ma esistono alcuni metodi che dovrebbero essere forniti da tutte le implementazioni.

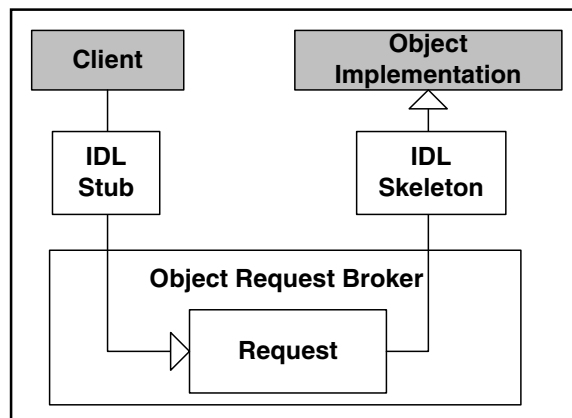
L'inizializzazione dell'ORB va effettuata invocando il metodo `init`

```
static ORB init()
static ORB init(Applet app, Properties props)
static ORB init(String[] args, Properties props)
```

Il metodo senza parametri opera secondo il pattern Singleton, restituendo un ORB di default con alcune limitazioni. Gli altri due metodi restituiscono un ORB con le proprietà specificate e sono pensati esplicitamente per le Java application e le Applet. L'array di stringhe e l'oggetto `Properties` consentono di impostare alcune proprietà dell'istanza di ORB restituita dal metodo `init`; l'array viene usato per i parametri da linea di comando. Le proprietà standard `ORBClass` e `ORBSingletonClass` consentono ad esempio di specificare l'utilizzo di un custom ORB differente da quello di default. Ogni implementazione fornisce anche proprietà aggiuntive; tutte le proprietà non riconosciute sono semplicemente ignorate.

Altre funzionalità sicuramente offerte da un ORB sono le operazioni relative agli object reference. Ogni riferimento a un oggetto (*Interoperable Object Reference*, IOR) può essere convertito in stringa; è garantito anche il processo inverso.

```
String object_to_string(Object obj)
Object string_to_object(String str)
```

Figura 16.2 – *Una richiesta da client a oggetto CORBA*

Prima che un oggetto remoto sia utilizzabile da un client, va attivato sull'ORB. Come si vedrà in seguito esistono più tipologie di attivazione. Il modo più semplice in assoluto è dato dai metodi

```
void connect(Object obj)
void disconnect(Object obj)
```

Il metodo `disconnect` disattiva l'oggetto consentendo al garbage collector di rimuoverlo.

In un contesto distribuito è necessario avere a disposizione meccanismi che consentano di scoprire quali oggetti CORBA sono disponibili e ottenere un riferimento ad essi. Anche in questo caso esistono più possibilità, la più semplice è fornita da due metodi dell'ORB

```
String[] list_initial_services()

Object resolve_initial_references(String object_name)
```

Il primo metodo elenca i servizi disponibili sull'ORB, mentre il secondo restituisce un generico riferimento a un oggetto individuato per nome. È bene precisare che un servizio è comunque un oggetto remoto e quindi recuperabile via `resolve_initial_references`.

Interoperabilità tra ORB

Le specifiche CORBA 1.1 si limitavano a dare le basi per la portabilità di oggetti applicativi e non garantivano affatto l'interoperabilità tra differenti implementazioni di ORB. Le specifiche 2.0 colmarono questa significativa lacuna con la definizione di un protocollo (GIOP) espressamente pensato per interazioni ORB-to-ORB.

Il General Inter-ORB Protocol specifica un insieme di formati di messaggi e di rappresentazioni dati comuni per la comunicazione tra ORB. I tipi di dato definiti da OMG sono mappati in un messaggio di rete flat (*Common Data Representation*, CDR).

GIOP definisce un formato multi-ORB di riferimento a un oggetto remoto, l'*Interoperable Object References* (IORs). L'informazione contenuta e specificata dalla struttura dello IOR assume significato indipendentemente dall'implementazione dell'ORB, consentendo a un'invocazione di transitare da un ORB a un altro. Ogni ORB fornisce un metodo `object_to_string` che consente di ottenere una rappresentazione stringa dello IOR di un generico oggetto.

Vista la diffusione di TCP/IP, comunemente viene usato l'Internet Inter-ORB Protocol (IIOP) che specifica come i messaggi GIOP vengono scambiati su TCP/IP. IIOP è considerato il protocollo standard CORBA e quindi ogni ORB deve connettersi con l'universo degli altri ORB traslando le request sul e dal backbone IIOP.

Tools e implementazioni CORBA

Per realizzare un'applicazione che utilizzi il middleware definito da OMG, occorre in primo luogo disporre di un prodotto che ne fornisca un'implementazione. La garanzia fornita è comunque quella di scrivere codice utilizzabile con differenti prodotti CORBA.

Lo standard CORBA è dinamico e complesso. Di conseguenza, lo scenario dei prodotti attualmente disponibili è in continuo divenire e il livello di aderenza dei singoli prodotti alle specifiche non è quasi mai completo. In ogni caso è sempre possibile utilizzare CORBA in maniera tale da garantire un'elevata portabilità.

Occorre comunque prestare molta attenzione alla scelta dell'ORB da utilizzare in quanto questi differiscono sia come prestazioni, sia come funzionalità fornite. Per una panoramica completa dei prodotti CORBA disponibili si veda [prodotti CORBA] in bibliografia.

Gli esempi presenti in questo capitolo fanno esplicito riferimento a due implementazioni: Sun Java IDL e Inprise VisiBroker. L'utilizzo di altri ORB con questi esempi potrebbe comportare modifiche.

Java IDL attualmente è disponibile in due versioni decisamente differenti. L'implementazione fornita con il JDK 1.2 è limitata e il compilatore IDL va scaricato a parte da `http://developer.java.sun.com/developer/earlyAccess/jdk12/idltojava.html`. Una migliore implementazione è presente invece nel Java 2 SDK a partire dalla versione 1.3.

Inprise VisiBroker è probabilmente la migliore e più diffusa implementazione CORBA presente sul mercato; è disponibile in versione trial: si veda [VisiBroker] in bibliografia.

Interface Definition Language

CORBA fornisce una chiara separazione tra l'interfaccia di un oggetto e la sua implementazione. In modo simile a quanto accade in RMI, il client non si deve occupare in modo diretto dei dettagli di implementazione, ma solo dell'interfaccia implementata dall'oggetto che intende utilizzare.

In un middleware distribuito tutti gli oggetti, compresi quelli che lo compongono, sono trattati come interfacce. Questo è sia una valida scelta di design, sia un'esigenza di distribuzione: un client tipicamente non conosce e non deve conoscere l'implementazione di un oggetto destinato a essere eseguito su una macchina server.

Questa considerazione ha una valenza ancora maggiore in un contesto tecnologico che consente ad esempio il dialogo tra oggetti Java e procedure Cobol che per natura probabilmente risiederanno addirittura su macchine ad architetture differenti.

Poiché CORBA è trasparente rispetto al linguaggio, OMG ha definito nelle sue specifiche un nuovo linguaggio interamente descrittivo (IDL) destinato alla definizione delle interfacce degli oggetti CORBA. In momenti successivi sono stati definiti i differenti mapping tra i vari linguaggi di programmazione e IDL. È da notare che in molti dei linguaggi utilizzabili con CORBA non esiste il concetto di interfaccia (ad esempio COBOL e C).

Un oggetto remoto quindi, indipendentemente dal fatto che sia applicativo o appartenente all'infrastruttura (l'ORB, i servizi, ecc.), per essere utilizzato in un middleware CORBA deve essere in primo luogo definito mediante IDL. Nel caso di un oggetto applicativo la definizione sarà a carico dello sviluppatore, nel caso di un oggetto di infrastruttura viene fornita da OMG. Ecco ad esempio parte della definizione IDL dell'ORB:

```
// IDL
module CORBA {
   interface ORB {
      string object_to_string (in Object obj);
      Object string_to_object (in string str);

Object resolve_initial_references (in ObjectId identifier) raises (InvalidName);

      // ecc...
   };
};
```

Sintassi e caratteristiche

La sintassi IDL è chiaramente C-like e quindi è piuttosto simile anche alla sintassi Java. Sebbene sia un linguaggio descrittivo orientato agli oggetti, in modo simile al C++, IDL include la possibilità, non contemplata da Java, di definire strutture dati che non siano classi.

I blocchi logici IDL sono racchiusi in parentesi graffe; a differenza di Java è necessario terminare sempre il blocco con un " `;` " e anche il singolo statement è terminato da un " `;` ". Con " `::` " è possibile specificare la gerarchia delle classi (equivale al " `.` " Java, per esempio `CORBA::Object`).

Nelle specifiche si parla di IDL come di un linguaggio case-insensitive, ma esistono implementazioni che non rispettano questa direttiva. A proposito delle regole di naming, va notato che CORBA non nasce nel mondo Java e quindi i tool IDL e le interfacce definite da OMG non rispettano le regole di naming abituali in un contesto Java.

In IDL è importante la sequenza delle definizioni dei vari elementi. Non è possibile utilizzare un elemento, sia esso una exception, una struttura dati o un'interfaccia, se non è già stato definito o almeno dichiarato; esiste comunque un meccanismo di forward declaration.

IDL non implementa l'override e l'overload, queste limitazioni sono legate al fatto che molti dei linguaggi supportati non forniscono queste caratteristiche. A differenza di quanto accade in Java, in un file IDL possono esistere molte interfacce pubbliche.

IDL in pratica

La definizione IDL di un oggetto permette di specificare solo gli aspetti relativi alla sua interfaccia. Si potranno quindi definire le signature dei metodi, le eccezioni che questi rilanciano, l'appartenenza ai package, costanti e strutture dati manipolate dai metodi.

Data la definizione IDL sarà necessario utilizzare un apposito compilatore fornito a corredo dell'ORB. Dalla compilazione si otterranno un buon numero di file `.java`, fra cui stub, skeleton e altri contenenti codice di supporto per l'aggancio all'ORB. A partire dai file generati sarà possibile realizzare l'opportuna implementazione Java.

Si provi a definire ad esempio una semplice costante in IDL

```
// IDL
module basic {
   const float PI = 3.14159;
};
```

Si compili il file IDL creato (nell'esempio `basic.idl`). Per la sintassi e il significato dei flag usati si rimanda alla documentazione dell'ORB.

`idltojava -fno-cpp basic.idl` (per l'implementazione JDK 1.2)

`idlj -fall basic.idl` (per l'implementazione J2SE 1.3)

`idl2java -boa basic.idl` (per l'implementazione VisiBroker)

Verrà creata una sottodirectory `basic` e un file `PI.java`

```
// JAVA
package basic;

public interface PI {
   public final static float value = (float)3.14159;
}
```

La generazione del file operata dal compilatore IDL è basata sulle regole di mapping definite da OMG per il linguaggio Java.

Mapping IDL–Java

La trasposizione da linguaggio IDL a linguaggio Java effettuata dal compilatore si basa sull'insieme di regole definite da OMG che costituiscono il mapping tra i due linguaggi.

Tipi base

La definizione di regole di mapping tra IDL e un linguaggio di programmazione implica in primo luogo la definizione di corrispondenze tra i differenti tipi di base; a runtime questo può causare errori di conversione durante il marshalling dei dati. La gestione di questi errori a runtime è a carico del programmatore.

Tabella 16.1 – *Corrispondenza tra tipi IDL e Java*

IDL Type	Java type	Exceptions
`boolean`	`boolean`	
`char`	`char`	`CORBA::DATA_CONVERSION`
`wchar`	`char`	`CORBA::DATA_CONVERSION`
`octet`	`byte`	
`string`	`java.lang.String`	`CORBA::MARSHAL` `CORBA::DATA_CONVERSION`
`wstring`	`java.lang.String`	`CORBA::MARSHAL` `CORBA::DATA_CONVERSION`
`short`	`short`	
`unsigned short`	`short`	
`long`	`int`	
`unsigned long`	`int`	
`long long`	`long`	
`unsigned long long`	`long`	
`float`	`float`	
`double`	`double`	
`fixed`	`java.math.BigDecimal`	`CORBA::DATA_CONVERSION`

Il problema si pone tipicamente per i tipi aventi precisione maggiore in Java che in IDL; ad esempio per i char che in Java, a differenza della maggior parte degli altri linguaggi, sono trattati come Unicode (16 bit) e non ASCII (8 bit). In IDL i tipi che trattano caratteri Unicode sono `wchar` e `wstring`. Alcuni dei tipi supportati da IDL non trovano corrispondenza in Java (ad esempio i tipi `unsigned`). `TRUE` e `FALSE` in IDL sono costanti e vengono mappate con i literal Java `true` e `false`.

Particolare attenzione va prestata all'utilizzo di `null`: CORBA non ha la nozione di `null` riferito alle stringhe o agli array. Un parametro stringa dovrà ad esempio essere trattato come una stringa vuota pena l'eccezione `org.omg.CORBA.BadParam`.

In IDL ogni tipo, base o complesso, può essere associato a un nome mediante la parola chiave `typedef`; poiché in Java il concetto di alias per un tipo non esiste, nel codice generato verranno usati comunque i tipi primitivi che lo compongono.

Module e interface

Come forse si è notato nell'esempio precedente la parola chiave `module` viene mappata esattamente sul package Java

```
// IDL
module basic {...}

// generated Java
package basic;
```

In IDL la keyword interface permette di specificare la vera e propria interfaccia dell'oggetto remoto definendone dati membro e metodi (nel gergo CORBA attributi e operazioni). Il mapping di un'interface è ottenuto con la generazione di un'interfaccia e alcune classi Java. Definendo la semplice interfaccia IDL

```
// IDL
module basic {

   interface HelloWorld {
      string hello();
   };

};
```

il compilatore creerà una directory `basic` e una serie di file Java (usando VisiBroker verranno generati anche altri file):

- `_HelloWorldImplBase` è lo skeleton, la classe base per la generazione dell'oggetto remoto; fornisce i meccanismi di ricezione di una request dall'ORB e quelli di risposta;
- `_HelloWorldStub` è lo stub, l'implementazione client-side dell'oggetto remoto; fornisce i meccanismi di conversione tra l'invocazione del metodo e l'invocazione via ORB dell'oggetto remoto;
- `HelloWorldOperations` è l'interfaccia Java che contiene le signature dei metodi;
- `HelloWorld` è l'interfaccia Java dell'oggetto remoto, specializza `HelloWorldOperations`;
- `HelloWorldHelper` e `HelloWorldHolder` saranno spiegati più avanti.

Insieme le interfacce `HelloWorldOperations` e `HelloWorld` definiscono l'interfaccia dell'oggetto CORBA; sono dette rispettivamente operations interface e signature interface. Il JDK 1.2 utilizza vecchie regole di mapping e non genera l'interfaccia operation.

La signature interface generata sarà

```
// generated Java
package basic;
public interface HelloWorld extends HelloWorldOperations, org.omg.CORBA.Object,
                                    org.omg.CORBA.portable.IDLEntity
}
```

mentre l'operations interface sarà

```
package basic;

public interface HelloWorldOperations {
   String hello ();
}
```

Come si vedrà più avanti, le altre classi serviranno come base per l'implementazione e l'utilizzo dell'oggetto remoto vero e proprio.

Il linguaggio IDL supporta l'ereditarietà multipla utilizzando la normale derivazione Java tra interfacce.

```
// IDL
module basic {

   interface ClasseBaseA {
      void metodoA();
   };
   interface ClasseBaseB {
      void metodoB();
   };
   interface ClasseDerivataAB: ClasseBaseA, ClasseBaseB {
   };

};
```

`ClasseDerivataAB` deriva dalle altre due interfacce e avrà quindi una rappresentazione Java.

```
// generated Java
package basic;
public interface ClasseDerivataAB extends ClasseDerivataABOperations,
                                          basic.ClasseBaseA, basic.ClasseBaseB {
}
```

un oggetto di questo tipo dovrà quindi fornire l'implementazione dei due metodi (`metodoA` e `metodoB`).

Attributi e metodi

In IDL le signature dei vari metodi sono fornite in maniera simile a Java. Per comodità è possibile dare una definizione dei metodi accessori di un attributo (i classici `get` e `set` Java) utilizzando la keyword attribute con l'eventuale modificatore `readonly`.

```
// IDL
module basic {

   interface Motocicletta {
      readonly attribute string colore;
      void cambiaMarcia(in long marcia);
   };

};
```

Poiché l'attributo colore è `readonly`, sarà generato solo il corrispondente metodo di lettura.

```
// generated Java
package basic;
public interface MotociclettaOperations {
   String colore();
   void cambiaMarcia(int marcia);
}
```

In IDL il passaggio di parametri a un metodo implica la dichiarazione del tipo di passaggio che si desidera adottare. Mentre in Java il passaggio per valore (tipi primitivi) o per riferimento (oggetti, array, ecc.) è implicitamente associato al tipo, in IDL è possibile specificarlo utilizzando nella signature le keyword `in`, `out` o `inout`. Come si può intuire un parametro `out` può essere modificato dal metodo invocato.

Poiché in Java non tutti i parametri sono trattati per riferimento, esistono delle classi wrapper apposite dette Holder.

Classi Holder

Le classi Holder sono utilizzate per supportare il passaggio di parametri `out` e `inout`. Come si è visto in precedenza, dalla compilazione di un'interfaccia IDL viene generata una corrispondente classe `<NomeInterfaccia>Holder`; l'Holder è generato per ogni tipo utente. Nel package `org.omg.CORBA` sono forniti gli Holder per tutti i tipi primitivi. Ogni Holder fornisce un costruttore di default che inizializza il contenuto a `false`, `0`, `null` o `null unicode` a seconda del tipo. Ecco per esempio l'Holder del tipo `base int`:

```
// JAVA
```

```
final public class IntHolder implements org.omg.CORBA.portable.Streamable {

   public int value;
   public IntHolder() {}
   public IntHolder(int initial) {...}

   public void _read(org.omg.CORBA.portable.InputStream is) {...}

   public void _write(org.omg.CORBA.portable.OutputStream os) {...}

   public org.omg.CORBA.TypeCode _type() {...}

}
```

Classi Helper

Per ogni tipo definito dall'utente il processo di compilazione genera una classe Helper con il nome `<TipoUtente>Helper`. La classe Helper è astratta e fornisce alcuni metodi statici che implementano varie funzionalità per manipolare il tipo associato (lettura e scrittura del tipo da/verso uno stream, lettura del repository id, e così via).

L'unica funzionalità di utilizzo comune è fornita dal metodo `narrow` implementato dall'Helper

```
// generated Java
package basic;
public class HelloWorldHelper {

   //...

   public static basic.HelloWorld narrow(
                  org.omg.CORBA.Object that) throws org.omg.CORBA.BAD_PARAM {

      if (that == null)
         return null;

      if (that instanceof basic.HelloWorld)
         return (basic.HelloWorld) that;

      if (!that._is_a(id())) {
         throw new org.omg.CORBA.BAD_PARAM();
      }

      org.omg.CORBA.portable.Delegate dup
      = ((org.omg.CORBA.portable.ObjectImpl) that)._get_delegate();

      basic.HelloWorld result = new basic._HelloWorldStub(dup);
```

```
        return result;
    }
}
```

Il metodo `narrow` effettua un cast "sicuro" dal generico Object Corba al tipo definito. Grazie a una serie di controlli ciò che verrà ritornato sarà sicuramente un oggetto del tipo atteso oppure una Exception `CORBA BAD_PARAM`.

Tipi strutturati

Come detto, mediante IDL è possibile dare la definizione di entità che non siano classi o interfacce, ma semplici strutture dati. Il mapping con Java sarà comunque operato mediante classi e interfacce.

Esistono tre categorie di tipi strutturati: `enum`, `union` e `struct`. Tutti i tipi strutturati sono mappati in Java con una final class fornita degli opportuni campi e costruttori, Helper e Holder.

L'`enum` è una lista ordinata di identificatori, la `union` è un incrocio tra la Union C e un'istruzione di switch, la `struct` è una struttura dati che consente di raggruppare al suo interno più campi.

```
// IDL
module basic {

    enum EnumType {first, second, third, fourth, fifth};

    union UnionType switch (EnumType) {
        case first: long win;
        case second: short place;
        default: boolean other;
    };

    struct Struttura {
        string campoA;
        string campoB;
    };

};
```

Nell'esempio, `Struttura` sarà mappata con Helper, Holder e la classe

```
// generated Java
package basic;
public final class Struttura implements org.omg.CORBA.portable.IDLEntity {
    // instance variables
```

```
   public String campoA;
   public String campoB;
   // constructors
   public Struttura() { }
   public Struttura(String __campoA, String __campoB) {
      campoA = __campoA;
      campoB = __campoB;
   }
}
```

Sequence e array

In IDL esistono due collezioni tipizzate di dati: sequence e array. Entrambe sono mappate su array Java. Le sequence possono avere dimensioni predefinite (bounded) o non predefinite (unbounded).

```
// IDL
module basic {

   typedef sequence<octet> ByteSequence;

   typedef string MioArray[20];

   struct StrutturaConArray {
      ByteSequence campoA;
   };
};
```

La compilazione dell'esempio genererà solo Helper e Holder per `ByteSequence` e `MioArray`. Nella struttura, il tipo `ByteSequence` sarà trattato come array di byte.

```
// generated Java
package basic;
public final class StrutturaConArray implements org.omg.CORBA.portable.IDLEntity {
   // instance variables
   public byte[] campoA;
   // constructors
   public StrutturaConArray() { }
   public StrutturaConArray(byte[] __campoA) {
      campoA = __campoA;
   }
}
```

Exception

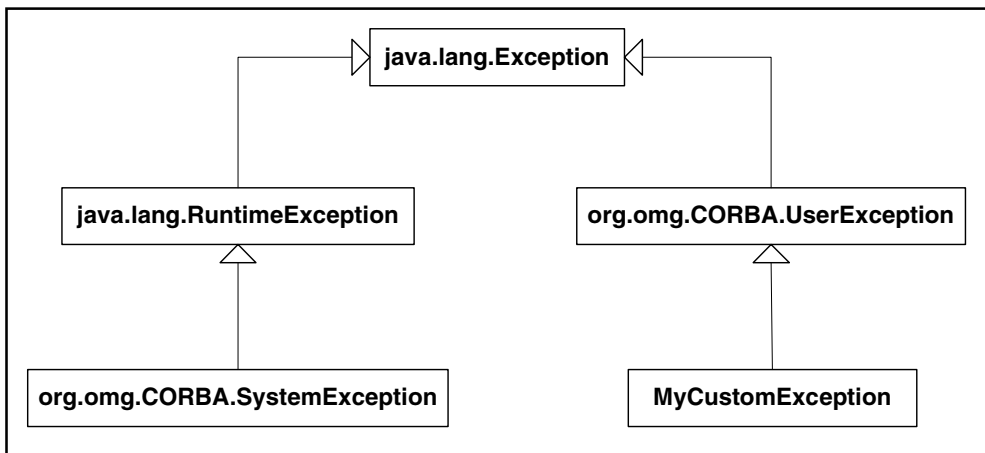
La definizione di una exception in IDL non è dissimile da quella di una struct. La

signature del metodo che la rilancia utilizza la keyword `raises` (equivalente del Java `throws`).

```
// IDL
module basic {

   exception MyCustomException {
      string reason;
   };
   interface HelloWorldWithException {
      string hello() raises (MyCustomException);
   };

};
```

Anche il mapping Java assomiglia a quello di una struct, quindi una classe final con i campi definiti nell'exception e i costruttori opportuni più i soliti Helper e Holder

```
// generated Java
package basic;
public final class MyCustomException extends org.omg.CORBA.UserException
                                    implements org.omg.CORBA.portable.IDLEntity {
   // instance variables
   public String reason;
   // constructors
   public MyCustomException() {
      super();
```

Figura 16.3 – *Gerarchia delle eccezioni CORBA*

```
   }
   public MyCustomException(String __reason) {
      super();
      reason = __reason;
   }
}
```

Le `SystemException` CORBA derivano da `java.lang.RuntimeException`, mentre ogni `UserException` definita in una IDL specializza `java.lang.Exception`. Per questa ragione è obbligatorio l'handle or declare su tutte le eccezioni utente, mentre non lo è per tutte le `SystemException` CORBA (`CORBA::MARSHAL`, `CORBA::OBJECT_NOT_EXIST`, ecc.).

Un po' di pratica

In un caso semplice i passi da seguire per creare, esporre e utilizzare un oggetto CORBA sono i seguenti:

- descrivere mediante IDL l'interfaccia dell'oggetto che si intende implementare;
- compilare con il tool apposito il file IDL;
- identificare tra le classi e le interfacce generate quelle necessarie alla definizione dell'oggetto e specializzarle opportunamente;
- scrivere il codice necessario per inizializzare l'ORB e informarlo circa la presenza dell'oggetto creato;
- compilare il tutto con un normale compilatore Java;
- avviare la classe di inizializzazione e l'applicazione distribuita.

Definizione IDL

Si definisca un semplice oggetto `Calcolatrice` che esponga un metodo in grado di computare la somma tra due numeri dati in input.

```
// IDL
module utility {

      interface Calcolatrice {

         long somma(in long a, in long b);
```

```
   };

};
```

Si compili il file `Calcolatrice.idl` mediante il compilatore fornito dall'ORB. Il processo di compilazione creerà una directory utility e gli opportuni file Java: `_CalcolatriceImplBase`, `_CalcolatriceStub`, `CalcolatriceOperations`, `Calcolatrice`, `CalcolatriceHelper` e `CalcolatriceHolder`.

Implementare l'oggetto remoto

La classe base per l'implementazione è la classe astratta `_CalcolatriceImplBase.java`, ovvero lo skeleton.

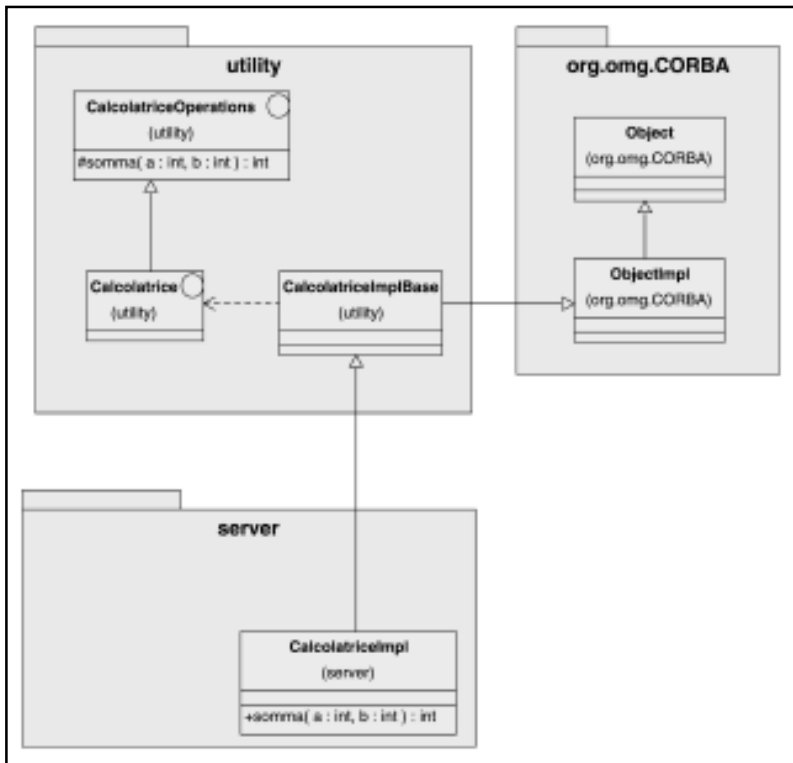
Si può notare nella fig. 16.4 come l'implementazione dell'interfaccia "remota" `Calcolatrice` non sia a carico dell'oggetto remoto, bensì a carico dello skeleton.

Lo skeleton è una classe astratta e non fornisce alcuna implementazione del metodo somma definito nell'interfaccia IDL. Quindi per definire il servant `CalcolatriceImpl` sarà necessario specializzare `_CalcolatriceImplBase` e fornire l'opportuna implementazione del metodo somma.

Ecco il codice completo del servant:

```
// JAVA
package server;

import utility.*;

public class CalcolatriceImpl extends _CalcolatriceImplBase {

   public CalcolatriceImpl() {
   super();
   }

   // Implementazione del metodo remoto
   public int somma(int a, int b) {
      return a + b;
   }
}
```

Poiché Java non supporta l'ereditarietà multipla, in alcune situazioni può essere limitante dover derivare necessariamente il servant da `ImplBase`. Nel caso in cui il servant debba derivare da un'altra classe è possibile utilizzare un meccanismo alternativo di delega detto Tie che non implica la specializzazione di `ImplBase`. In questo capitolo l'approccio Tie non sarà esaminato.

Figura 16.4 – *Gerarchia di derivazione della classe servant*

Implementare la classe Server

Si è già avuto modo di notare come nel gergo CORBA il componente remoto che espone i servizi venga definito servant. Il server invece è la classe che inizializza l'environment, istanzia l'oggetto remoto, lo rende disponibile ai client e si pone in attesa.

La classe server è quindi una classe di servizio che ha come compito fondamentale quello di creare e agganciare all'ORB l'istanza di oggetto remoto che utilizzeranno i client e di fornire a questa un contesto di esecuzione.

L'inizializzazione dell'ORB è effettuata utilizzando il metodo `init`.

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
```

Il parametro `args` è semplicemente l'array di input. Saranno quindi valorizzabili da linea di comando alcune proprietà dell'ORB (per un elenco delle proprietà disponibili consultare la documentazione dell'implementazione CORBA utilizzata).

In questo primo esempio l'aggancio è effettuato senza l'ausilio di un Object Adapter. Come già anticipato, una semplice forma di "registrazione" è fornita dal metodo `connect` dell'ORB

```
CalcolatriceImpl calc = new CalcolatriceImpl();
orb.connect(calc);
```

Utilizzando un meccanismo di questo tipo, il servant va considerato come un oggetto CORBA di tipo transient. Un riferimento a un oggetto di questo tipo è valido solo nel tempo di vita di una precisa istanza del servant. Più avanti saranno analizzati gli oggetti di tipo persistent.

Per ogni ORB CORBA 2.0 compliant, l'object reference (IOR) in versione stringa è ottenibile invocando

```
orb.object_to_string(calc)
```

La stringa ottenuta è il riferimento CORBA all'istanza di calcolatrice; come tale è esattamente tutto ciò di cui necessita un client per accedere ai servizi dell'oggetto. Per fornire al client lo IOR esistono molte soluzioni, la più semplice consiste nel salvarlo su file.

```
PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(args[0])));
out.println(orb.object_to_string(calc));
out.flush();
out.close();
```

A questo punto il server può mettersi in attesa. L'attesa è necessaria in quanto l'istanza di calcolatrice "vive" solo e soltanto nel contesto fornito dall'applicazione server; è all'interno di questa che è stato effettuato il new. Si può implementare un'attesa idle del processo server utilizzando il metodo `wait` di un Java Object

```
java.lang.Object sync = new java.lang.Object();
synchronized (sync) {
   sync.wait();
}
```

Ecco il codice completo della classe `CalcolatriceServer`.

```
// JAVA
package server;

import utility.*;
import java.io.*;

public class CalcolatriceServer {
```

```
   public static void main(String[] args) {

      if (args.length!=1) {
         System.err.println("Manca argomento: path file ior");
         return;
      }

      try {

         // Inizializza l'ORB.
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

         // Crea un oggetto Calcolatrice
         CalcolatriceImpl calc = new CalcolatriceImpl();
         orb.connect(calc);

         // Stampa l'object reference in versione stringa
         System.out.println("Creata Calcolatrice:\n"
                            + orb.object_to_string(calc));

         // Scrive l'object reference nel file
         PrintWriter out
         = new PrintWriter(new BufferedWriter(new FileWriter(args[0])));
         out.println(orb.object_to_string(calc));
         out.close();

         // Attende l'invocazione di un client
         java.lang.Object sync = new java.lang.Object();
         synchronized (sync) {
            sync.wait();
         }

      }
      catch (Exception e) {
         System.err.println("Server error: " + e);
         e.printStackTrace(System.out);
      }
   }
}
```

Implementare il Client

Il client dovrà in primo luogo inizializzare l'ORB con il metodo `init()`, come effettuato nella classe server.

Per ottenere il riferimento all'istanza di calcolatrice, il client dovrà leggere lo IOR memorizzato nel file generato dal server.

```
BufferedReader in = new BufferedReader(new  FileReader(args[0]));
```

```
String ior = in.readLine();
in.close();
```

È possibile ottenere lo IOR invocando il metodo opposto a quello utilizzato per trasformarlo in stringa. Il metodo `string_to_object` fornito dall'ORB restituisce un CORBA Object a partire dalla stringa che rappresenta il suo IOR.

```
org.omg.CORBA.Object obj = orb.string_to_object(ior);
```

Il metodo `string_to_object` restituisce un oggetto di tipo generico e non un riferimento che consenta di invocare il metodo somma.

Per ottenere tale riferimento, in uno scenario Java, si effettuerebbe un cast (in RMI, ad esempio, dopo aver effettuato una lookup si opera un cast per ottenere il tipo corretto). In un contesto CORBA invece, per convertire il generico oggetto in un oggetto di tipo determinato, bisogna utilizzare il metodo `narrow` della classe `<Tipo>Helper`.

```
Calcolatrice calc = CalcolatriceHelper.narrow(obj);
```

A questo punto il client è in condizione di invocare il metodo remoto con le medesime modalità usate per una comune invocazione di metodo

```
calc.somma(a, b)
```

dove `a` e `b` sono da intendersi come 2 int. È da notare come questo non sia l'unico modello di invocazione CORBA. Un'invocazione di questo tipo viene detta invocazione statica, più avanti sarà affrontata l'invocazione dinamica.

Ecco il codice completo del client:

```
package client;

import utility.*;
import java.io.*;

public class CalcolatriceClient {

   public static void main(String args[]) {

      if (args.length!=1) {
         System.err.println("Manca argomento: path file ior");
         return;
      }

      try {
```

```
         // Crea e inizializza l'ORB
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

         // Legge dal file il reference all'oggetto
         // Si assume che il server lo abbia generato
         BufferedReader in = new BufferedReader(new FileReader(args[0]));
         String ior = in.readLine();
         in.close();

         // Ottiene dal reference un oggetto remoto...
         org.omg.CORBA.Object obj = orb.string_to_object(ior);

         // ...e ne effettua il narrow a tipo Calcolatrice
         Calcolatrice calc = CalcolatriceHelper.narrow(obj);

         // Ottiene da input tastiera i 2 numeri da sommare
         BufferedReader inputUser
         = new BufferedReader(new InputStreamReader(System.in));
         String first, second;
         int a, b;

         // Leggo primo addendo
         System.out.println ();
         System.out.print("A = ");
         first = inputUser.readLine();
         a = Integer.valueOf(first).intValue ();

         // Leggo secondo addendo
         System.out.println ();
         System.out.print("B = ");
         second = inputUser.readLine();
         b = Integer.valueOf(second).intValue ();

         // Invoca il metodo remoto passandogli i parametri
         System.out.println ();
         System.out.print("Il risultato è: ");
         System.out.print(calc.somma(a, b));

      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Eseguire l’esempio

Dopo aver compilato tutte le classi, comprese quelle generate dal precompilatore, è finalmente possibile eseguire l’esempio.

La prima classe da mandare in esecuzione è la classe `CalcolatriceServer`

```
java server.CalcolatriceServer calc.ior
```

passando come parametro il path del file su cui si intende memorizzare lo IOR. L'output prodotto sarà

```
Creata Calcolatrice:
IOR:000000000000001d49444c3a7574696c6974792f43616c636f6c6174726963653a312e
300000000000000001000000000000002c0001000000000004696e6b0005b7000000000018afab
cafe00000002a1ed120b000000080000000000000000
```

Si noti che, poiché il processo server si pone in attesa, l'esecuzione effettivamente non termina. Se si terminasse il processo, il client avrebbe uno IOR inservibile in quanto l'istanza identificata da questo non sarebbe più "viva".

A questo punto si può mandare in esecuzione il client fornendogli il path del file generato dal server

```
java client.CalcolatriceClient calc.ior
```

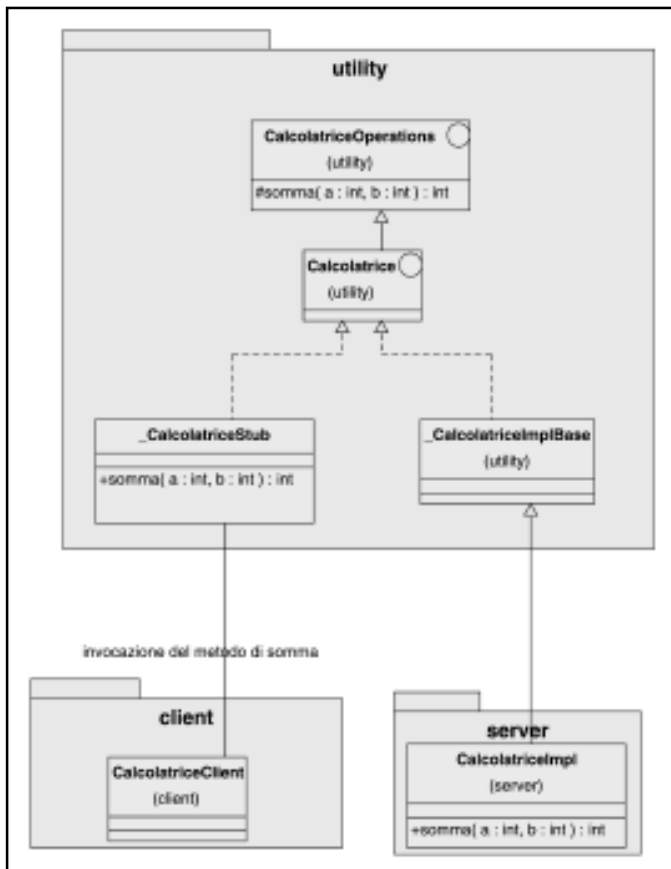
Il programma richiederà in input i due numeri da sommare (si noti che non sono stati gestiti eventuali errori di conversione) e stamperà, a seguito di un'invocazione remota, il risultato.

Client e server stub

È bene effettuare una breve digressione sul concetto di stub (letteralmente "surrogato"). Lo stub compare in molti scenari di programmazione (ad esempio in DLL e in generale nella programmazione distribuita). Esiste una precisa corrispondenza con un celebre pattern di programmazione: il pattern Proxy.

Il pattern Proxy viene tipicamente utilizzato per aggiungere un livello di indirezione. Il Proxy è un surrogato di un altro oggetto ed è destinato a controllare l'accesso a quest'ultimo. In generale implica la presenza di un oggetto, detto Proxy, che abbia la stessa interfaccia dell'oggetto effettivo, detto Real Subject. Il Proxy riceve le richieste destinate al Real Subject e le comunica a quest'ultimo effettuando eventualmente delle operazioni prima e/o dopo l'accesso.

Osservando la fig. 16.5 è possibile vedere come lo stub e lo skeleton implementino la stessa interfaccia, quella dell'oggetto remoto. Quindi un generico client sarà in grado di dialogare con lo stub invocando i metodi che intende far eseguire all'oggetto remoto. In questo senso lo stub opera da procuratore dell'oggetto presso il client (*proxy* significa per l'appunto "procuratore", "delegato").

Figura 16.5 – *Il pattern Proxy e gli stub server e client*

Lo stub, nella sua opera di delegato, sarà in grado di rendere invisibili al client tutti i dettagli della comunicazione remota e della locazione fisica dell'oggetto. Nell'ottica del client il dialogo sarà operato direttamente con l'oggetto remoto (questo è garantito dal fatto che lo stub implementa l'interfaccia dell'oggetto remoto). Sostituire un'implementazione non avrà alcun impatto sul client purché l'interfaccia rimanga inalterata.

Un possibile miglioramento

La soluzione proposta nell'esempio precedente è decisamente primitiva. Di fatto l'accesso a un oggetto remoto è possibile solo se il client e il server condividono una porzione di file system (in realtà sono utilizzabili anche altri meccanismi quali mail, floppy, …).

Un primo miglioramento potrebbe essere ottenuto utilizzando un Web Server. Con un Web Server attivo un client potrebbe leggere il file contenente lo IOR via HTTP. Il codice di lettura del client potrebbe essere qualcosa del genere:

```
URL urlIOR = new URL(args[0]);
DataInputStream in = new DataInputStream(urlIOR.openStream());
String ior = in.readLine();
in.close();
```

Al client andrebbe fornito non più il path, ma l'URL corrispondente al file generato dal server, ad esempio `http://localhost/corba/calc.ior`.

Il server dovrà generare il file nella virtual directory corretta del Web Server (`corba` nell'esempio). Nel caso non si disponga di un Web Server è possibile scaricare gratuitamente lo "storico" Apache da `www.apache.org`.

Anche con questa modifica la soluzione, pur essendo praticabile in remoto e totalmente portabile, è ben lontana dall'ottimale. Tra i difetti che presenta è bene notare come in parte violi la location transparency promessa da CORBA: non si conosce l'effettiva collocazione dell'implementazione, ma è necessario conoscere l'URL del file IOR.

Un approccio decisamente migliore prevede l'utilizzo del CORBA Naming Service.

CORBA Naming Service

Il Naming Service è sicuramente il principale meccanismo CORBA per la localizzazione di oggetti su un ORB. Fa parte delle specifiche CORBA dal 1993 ed è il servizio più importante tra quelli standardizzati da OMG.

Fornisce un meccanismo di mapping tra un nome e un object reference, quindi rappresenta anche un metodo per rendere disponibile un servant a un client remoto. Il meccanismo è simile a quello di registrazione e interrogazione del registry in RMI.

Nella programmazione distribuita l'utilizzo di un nome per reperire una risorsa ha degli evidenti vantaggi rispetto all'utilizzo di un riferimento. In primo luogo il nome è significativo per lo sviluppatore, in secondo luogo è completamente indipendente dagli eventuali restart dell'oggetto remoto.

Struttura del Naming Service

L'idea base del Naming Service è quella di incapsulare in modo trasparente i servizi di naming e directory già esistenti. I nomi quindi sono strutturabili secondo uno schema gerarchico ad albero, uno schema astratto indipendente dalle singole convenzioni delle varie piattaforme di naming o di directory. L'operazione che associa un nome a un reference è detta bind (esiste pure l'operazione di unbind). L'operazione che recupera un reference a partire da un nome è detta naming resolution.

Esistono dei naming context all'interno dei quali il nome è univoco. I naming context possono essere ricondotti ai nodi intermedi dell'albero di naming e al concetto di directory in un file system. Possono esistere più nomi associati a uno stesso oggetto.

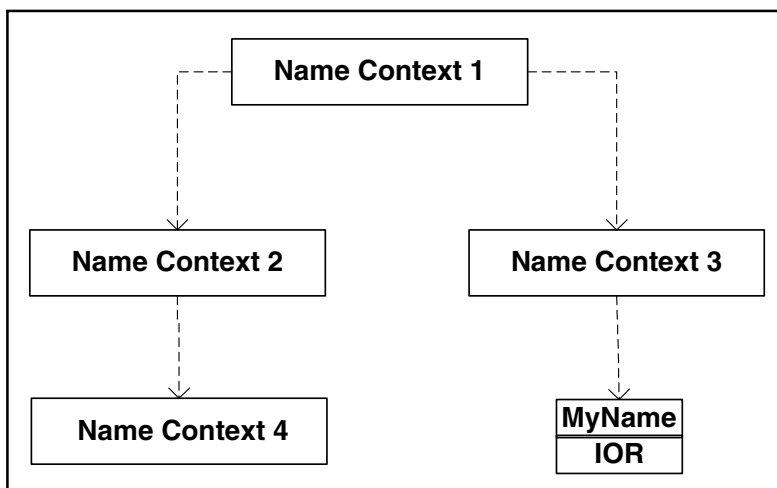
In questo scenario quindi un nome è una sequenza di name components; questi formano il cosiddetto compound name. I nodi intermedi sono utilizzati per individuare un context, mentre i nodi foglia sono i simple name.

Un compound name quindi individua il cammino (path) che, attraverso la risoluzione di tutti i context, porta al simple name che identifica la risorsa.

Ogni `NameComponent` è una struttura con due elementi. L'identifier è la stringa nome, mentre il kind è un attributo associabile al component. Questo attributo non è considerato dal Naming Service, ma è destinato al software applicativo.

La definizione IDL del `NameComponent` è la seguente

```
// IDL
module CosNaming {

   struct NameComponent {
      Istring id;
      Istring kind;
   };

   typedef sequence <NameComponent> Name;
}
```

Figura 16.6 – *Struttura ad albero di Naming*

L'interfaccia principale è `NamingContext` e fornisce tutte le operazioni necessarie alla definizione dell'albero di Naming e alla sua navigazione. Per quel che concerne il bind vengono fornite due funzionalità di bind tra Name – Object, due funzionalità di bind Name – Naming Context e una funzionalità di unbind.

```
// IDL
module CosNaming{

   // ...

   interface NamingContext
   {

      // ...

      void bind(in Name n, in Object obj) raises(NotFound, CannotProceed,
                                                 InvalidName, AlreadyBound);

      void rebind(in Name n, in Object obj) raises(NotFound, CannotProceed,
                                                   InvalidName);

      void bind_context(in Name n, in NamingContext nc) raises(NotFound, CannotProceed,
                                                               InvalidName,
                                                               AlreadyBound);

      void rebind_context(in Name n, in NamingContext nc) raises(NotFound, CannotProceed,
                                                                 InvalidName);

      void unbind(in Name n) raises(NotFound, CannotProceed, InvalidName);

   };
};
```

I metodi rebind differiscono dai metodi bind semplicemente nel caso in cui il nome sia già presente nel context; `rebind` sostituisce l'object reference mentre `bind` rilancia un'eccezione `AlreadyBound`.

La definizione dell'albero implica anche la possibilità di creare o distruggere i `NamingContext`.

```
// IDL
module CosNaming{

   // ...

   interface NamingContext
```

```
   {

      // ...

      NamingContext new_context();

      NamingContext bind_new_context(in Name n) raises(NotFound, AlreadyBound,
                                                       CannotProceed,
                                                       InvalidName);

      void destroy() raises(NotEmpty);

   };
};
```

La risoluzione di un name è attuata mediante il metodo `resolve`. La procedura di risoluzione di un nome gerarchico implicherà la navigazione ricorsiva dell'albero di context.

```
// IDL
module CosNaming{

   // ...

   interface NamingContext
   {

      // ...

      Object resolve(in Name n) raises(NotFound, CannotProceed, InvalidName);

   };
};
```

La navigazione del Naming è invece legata all'utilizzo del metodo `list` che ritorna un insieme di name sui quali è possibile operare iterativamente. Più precisamente ciò che viene ritornato è un oggetto di tipo `BindingIterator` che, tramite i metodi `next_one` e `next_n`, permette di navigare attraverso tutti i bind associati al context.

```
// IDL
module CosNaming{

   //...

   interface BindingIterator {

      boolean next_one(out Binding b);
```

```
      boolean next_n(in unsigned long how_many, out BindingList bl);

      void destroy();
   };

   interface NamingContext
   {

      // ...

      void list(in unsigned long how_many, out BindingList bl,
                out BindingIterator bi);

   };
};
```

Utilizzare il Naming Service

Alla luce di quanto visto finora sul Naming Service, è possibile migliorare l'esempio della calcolatrice visto in precedenza. Lo scenario generale non cambia: esiste un oggetto servant (non necessita di alcuna modifica), un oggetto server di servizio (pubblica il servant) e il client.

Il Naming Service ha impatto solo sulle modalità di registrazione e di accesso all'oggetto, quindi solo sul client e sul server. Anche l'IDL non necessita di alcuna modifica.

Per pubblicare un oggetto, il server dovrà in primo luogo ottenere un riferimento al Naming Service. Come detto in precedenza, è possibile ottenere un riferimento a un qualunque servizio CORBA invocando sull'ORB il metodo `resolve_initial_references`. Ottenuto il riferimento, come per qualunque oggetto CORBA, andrà utilizzato il `narrow` fornito dal corrispondente Helper.

```
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

A questo punto è possibile effettuare il bind tra il `NameComponent` opportuno e l'istanza di servant (`calc` è in questo caso l'istanza di `CalcolatriceImpl`; si veda più avanti il codice completo).

```
NameComponent nc = new NameComponent("Calc", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, calc);
```

I due parametri del costruttore `NameComponent` sono, rispettivamente, il name e il kind.

Ecco il codice completo della classe server

```
package server;

import utility.*;
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

public class CalcolatriceServer {

   public static void main(String[] args) {

      try {

         // Inizializza l'ORB.
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

         // Crea un oggetto Calcolatrice
         CalcolatriceImpl calc = new CalcolatriceImpl();
         orb.connect(calc);

         // Stampa l'object reference in versione stringa
         System.out.println("Creata Calcolatrice:\n"
                              + orb.object_to_string(calc));

         // Root naming context
         org.omg.CORBA.Object objRef
         = orb.resolve_initial_references("NameService");

         NamingContext ncRef = NamingContextHelper.narrow(objRef);

         // Fa il bind dell'oggetto nel Naming
         NameComponent nc = new NameComponent("Calc", " ");
         NameComponent path[] = {nc};
         ncRef.rebind(path, calc);

         // Attende l'invocazione di un client
         java.lang.Object sync = new java.lang.Object();
         synchronized (sync) {
            sync.wait();
         }

      } catch (Exception e) {
         System.err.println("Server error: " + e);
         e.printStackTrace(System.out);
      }
   }
}
```

Il client dovrà effettuare le stesse operazioni per ottenere il riferimento al Naming Service. Poi dovrà effettuare la resolve per ottenere un riferimento all'oggetto remoto `Calcolatrice`.

```
NameComponent nc = new NameComponent("Calc", " ");
NameComponent path[] = {nc};
Calcolatrice calc = CalcolatriceHelper.narrow(ncRef.resolve(path));
```

Ecco il codice completo della classe client.

```
package client;

import utility.*;
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CalcolatriceClient {

   public static void main(String args[]) {

      try {

         // Crea e inizializza l'ORB
         ORB orb = ORB.init(args, null);

         // Root naming context
         org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");

         NamingContext ncRef = NamingContextHelper.narrow(objRef);

         // Utilizzo il Naming per ottenere il riferimento all'oggetto
         NameComponent nc = new NameComponent("Calc", "");
         NameComponent path[] = {nc};
         Calcolatrice calc = CalcolatriceHelper.narrow(ncRef.resolve(path));

         // Ottiene da input tastiera i 2 numeri da sommare
         BufferedReader inputUser
         = new BufferedReader (new InputStreamReader(System.in));
         String first, second;
         int a, b;

         // Leggo primo addendo
         System.out.println ();
         System.out.print("A = ");
         first = inputUser.readLine();
         a = Integer.valueOf(first).intValue ();
```

```
         // Leggo secondo addendo
         System.out.println ();
         System.out.print("B = ");
         second = inputUser.readLine();
         b = Integer.valueOf(second).intValue ();

         // Invoca il metodo remoto passandogli i parametri
         System.out.println ();
         System.out.print("Il risultato è: ");
         System.out.print(calc.somma(a, b));

      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Per eseguire l'esempio sarà necessario effettuare un passo in più rispetto a quanto fatto in precedenza, ovvero attivare il Naming Service prima di mandare in esecuzione il server.

L'attivazione del servizio è legata all'implementazione CORBA, sarà quindi differente da ORB a ORB (nel caso si utilizzi VisiBroker sarà necessario avviare anche il tool OsAgent trattato in seguito).

```
tnameserv -ORBInitialPort 1050
```
(per l'implementazione Sun)

```
nameserv NameService
```
(per l'implementazione VisiBroker)

A questo punto sarà possibile avviare il server (per il significato dei flag utilizzati si rimanda alla documentazione del prodotto).

```
java server.CalcolatriceServer -ORBInitialPort 1050
```
(per l'implementazione Sun)

```
java -DSVCnameroot
=NameService server.CalcolatriceServer
```
(per l'implementazione VisiBroker)

E infine avviare il client

```
java client.CalcolatriceClient -ORBInitialPort 1050
```
(per l'implementazione Sun)

```
java -DSVCnameroot
=NameService client.CalcolatriceClient
```
(per l'implementazione VisiBroker)

È possibile notare come in questa modalità l'utilizzo di CORBA abbia parecchie similitudini con quello di RMI: l'utilizzo di `tnameserv` (`nameserv`) rimanda a quello di `rmiregistry`, così come i metodi del `NamingContext` `bind` e `rebind` rimandano ai

metodi usati per RMI `Naming.bind` e `Naming.rebind`. Una volta ottenuto il reference, l'utilizzo dell'oggetto remoto è sostanzialmente identico.

Accesso concorrente a oggetti remoti

Uno dei compiti più complessi nell'ambito della programmazione distribuita è l'implementazione e la gestione della concorrenza. Un oggetto remoto distribuito sulla rete può essere utilizzato contemporaneamente da più client. Questo è uno dei fattori che rendono allettante la programmazione distribuita poiché l'utilizzo in concorrenza di una qualunque risorsa implica un suo migliore sfruttamento.

La gestione della concorrenza in ambiente distribuito è strettamente legata al design dell'applicazione e tipicamente va a ricadere in uno dei tre approcci qui riportati.

- unico thread: il thread è unico, le richieste sono gestite in modo sequenziale ed eventualmente accodate;
- un thread per client: viene associato un thread alla connessione con il client; le successive richieste del client sono a carico di questo thread;
- un thread per request: esiste un pool di thread utilizzati in modo concorrente per rispondere alle richieste dei client.

Esisteranno comunque situazioni in cui i differenti thread dovranno accedere a una risorsa condivisa (p.e.: connessione a DB, file di log, …); in questi casi l'accesso alla risorsa andrà opportunamente sincronizzato.

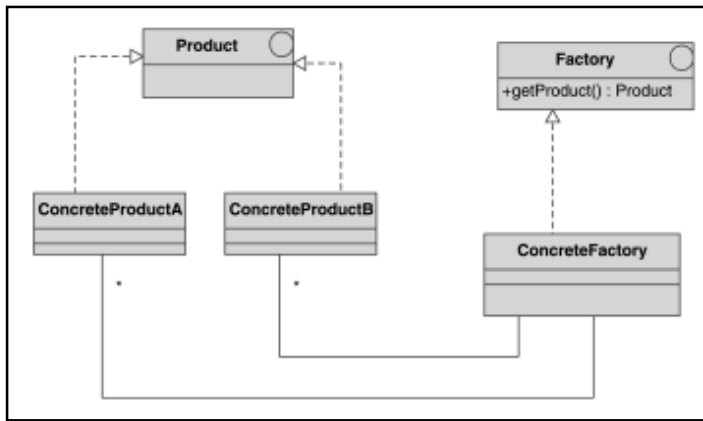
Come si vedrà più avanti, con CORBA è possibile specificare le politiche di gestione dei thread mediante l'utilizzo degli object adapter. Nella pratica, questi aspetti vengono spesso gestiti applicativamente con l'adozione di opportuni pattern di programmazione.

Questo permette di realizzare soluzioni complesse del tutto indipendenti dall'implementazione e dalla tecnologia. Tipicamente queste soluzioni sono attuate con l'adozione del pattern Factory.

Il pattern Factory

Quando si istanzia un oggetto è necessario fare un riferimento diretto ed esplicito a una precisa classe, fornendo gli eventuali parametri richiesti dal costruttore. Ciò vincola implicitamente l'oggetto utilizzatore all'oggetto utilizzato. Se questo può essere accettabile nella maggior parte dei casi, talvolta è un limite troppo forte.

In questi casi può essere molto utile incapsulare in una classe specializzata tutte le valutazioni relative alla creazione dell'oggetto che si intende utilizzare. Questa soluzione

Figura 16.7 – *Il pattern Factory*

è il più noto tra i creational patterns: il pattern Factory (anche detto Factory Method o Virtual Constructor). *Factory* letteralmente vuol dire "fabbrica" ed è proprio questo il senso dell'oggetto `Factory`: fabbricare sulla base di alcune valutazioni un determinato oggetto. Più formalmente, facendo riferimento anche alla fig. 16.7, un oggetto `Factory` fabbrica oggetti `ConcreteProduct` appartenenti a una determinata famiglia specificata dalla sua interfaccia (o classe astratta) `Product`.

Un client non crea mai in modo diretto un'istanza del `Product` (in un contesto remoto probabilmente non ne conoscerà nemmeno la classe), ma ne ottiene un'istanza valida attraverso l'invocazione del `FactoryMethod` sull'oggetto `Factory`. In questo modo è possibile sostituire in ogni momento l'implementazione `ConcreteProductA` con un'implementazione omologa `ConcreteProductB` senza che un eventuale client se ne accorga.

In realtà i vantaggi elencati in precedenza sono già impliciti in uno scenario CORBA (il disaccoppiamento è garantito dalla funzionalità Proxy dello stub). Nella programmazione distribuita il pattern Factory ha però altri vantaggi, in particolare consente di implementare meccanismi di load-balancing e fault-tolerance.

Poiché è la Factory a determinare la creazione del Product, essa potrà:

- istanziare un oggetto per ogni client;
- applicare un round-robin tra le differenti istanze già create, ottenendo un semplice load-balancing;
- restituire differenti tipologie di oggetti appartenenti alla famiglia Product sulla base di valutazioni legate all'identità del client;

- implementare un semplice fault-tolerance, escludendo dal pool di oggetti quelli non più funzionanti o non più raggiungibili via rete.

Un esempio di Factory

Per sperimentare un tipico caso di applicazione del pattern Factory si realizzi il classico "carrello della spesa". Per l'esempio saranno implementate la classe carrello (`ShoppingCart`) e una sua Factory (`ShoppingCartFactory`). La prima fornirà una funzionalità di acquisto e una di restituzione del contenuto, la seconda sarà dotata del solo metodo `getShoppingCart`.

Si definisce l'IDL

```
// IDL
module shopping {

   struct Book {
      string Author;
      string Title;
   };

   typedef sequence <Book> BookList;

   interface ShoppingCart {
      void addBook(in Book book);
      BookList getBookList();
   };

   interface ShoppingCartFactory {
      ShoppingCart getShoppingCart(in string userID);
   };
};
```

Sono definite come interface sia la Factory (con il Factory Method `getShoppingCart`), sia il carrello vero e proprio. La Factory può essere considerata una classe di infrastruttura, mentre tutti i metodi di business sono implementati nello `ShoppingCart`.

La classe `ShoppingCartImpl` implementa i due semplici metodi di business e non presenta nulla di nuovo rispetto a quanto visto in precedenza.

```
package server;

import shopping.*;
import java.util.Vector;

public class ShoppingCartImpl extends _ShoppingCartImplBase {
```

```
   Vector v = new Vector();

   public ShoppingCartImpl() {
      super();
   }

   // Aggiunge un libro al carrello
   public void addBook(Book book) {
      v.add(book);
   }

   // Restituisce l'elenco dei libri acquistati
   public Book[] getBookList() {
      Book[] books = new Book[v.size()];

      for (int i=0; i<v.size(); i++)
         books[i] = (Book) v.elementAt(i);

      return books;
   }
}
```

Più interessante è l'oggetto `Factory` che ha il compito di generare le istanze di `ShoppingCartImpl` da assegnare ai client. Nel metodo `getShoppingCart` viene stabilita la politica di creazione e restituzione delle istanze di carrello, nel caso in esame la decisione è ovvia in quanto il carrello ha evidentemente un rapporto uno a uno con i client.

Per memorizzare le varie istanze viene utilizzato un oggetto di tipo `Dictionary`. Alla prima connessione dell'utente la `Factory` creerà il carrello e lo "registrerà" sull'ORB. La `Factory` può ottenere un riferimento valido all'ORB invocando il metodo `_orb()` fornito da `org.omg.CORBA.Object`.

Ecco la classe `Factory`.

```
package server;

import shopping.*;
import java.util.*;

public class ShoppingCartFactoryImpl extends _ShoppingCartFactoryImplBase {

   private Dictionary allCarts = new Hashtable();

   public ShoppingCartFactoryImpl() {

      super();
   }

   public synchronized ShoppingCart getShoppingCart(String userID) {
```

```
      // Cerca il carrello assegnato allo userID...
      shopping.ShoppingCart cart
      = (shopping.ShoppingCart) allCarts.get(userID);

      // ...se non lo trova...
      if(cart == null) {

         // Crea un nuovo carrello...
         cart = new ShoppingCartImpl();

         // ...e lo attiva sull'ORB
         _orb().connect(cart);

         System.out.println("Created " + userID + "'s cart: " + cart);

         // Salva nel dictionary associandolo allo userID
         allCarts.put(userID, cart);
      }

      // Restituisce il carrello
      return cart;
   }
}
```

È da notare che la `Factory` sarà utilizzata in concorrenza da più client, di conseguenza sarà opportuno sincronizzare il metodo `getShoppingCart` per ottenere un'esecuzione consistente.

Per quanto detto in precedenza, un client otterrà un oggetto remoto di tipo `ShoppingCart` interagendo con la `Factory`. Pertanto, l'unico oggetto registrato sul Naming Service sarà l'oggetto `Factory`. La registrazione sarà effettuata dalla classe server con il nome `ShoppingCartFactory` con le stesse modalità viste nell'esempio precedente (il codice non viene qui mostrato).

Dopo aver ottenuto il reference allo `ShoppingCart` assegnato, il client potrà operare direttamente su questo senza interagire ulteriormente con la `Factory`.

```
package client;

import shopping.*;
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ShoppingCartClient {

   public static void main(String args[]) {

      if (args.length != 3) {
```

```
         System.err.println("Uso corretto: java ShoppingCartClient
                            userId Autore Titolo");
         return;
      }

      try{

         // Crea e inizializza l'ORB
         ORB orb = ORB.init(args, null);

         // Root naming context
         org.omg.CORBA.Object objRef
         = orb.resolve_initial_references("NameService");

         NamingContext ncRef = NamingContextHelper.narrow(objRef);

         // Utilizzo il Naming per ottenere il
         // riferimento all'oggetto Factory
         NameComponent nc = new NameComponent("ShoppingCartFactory", " ");
         NameComponent path[] = {nc};
         ShoppingCartFactory factory
         = ShoppingCartFactoryHelper.narrow(ncRef.resolve(path));

         // Ottengo dalla Factory un'oggetto ShoppingCart
         ShoppingCart cart = factory.getShoppingCart(args[0]);

         // Aggiungo un libro
         cart.addBook(new Book(args[1],args[2]));

         // Ottengo la lista dei libri e la stampo
         Book[] list = cart.getBookList();

         for(int i=0; i<list.length; i++)
            System.out.println("Autore " + list[i].Author
                               + " - Titolo " + list[i].Title);

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

I passi necessari per l’esecuzione sono gli stessi visti nell’esempio precedente. Al client vanno “passati” lo userId, l’autore e il titolo del libro da aggiungere al carrello.

L’adozione di un pattern è una scelta del tutto indipendente dalla tecnologia e quindi adattabile a qualunque altro scenario (ad esempio RMI). Un design come quello visto rende l’architettura più robusta e adattabile, consentendo di modificare e configurare il

comportamento anche a start up avvenuto senza alcun impatto sui client. In ottica Enterprise la resistenza ai cambiamenti è di massimo interesse.

Come già accennato, le politiche relative alla gestione delle istanze sono configurabili anche utilizzando gli Object Adapter.

Utilizzo degli Object Adapter

L'Object Adapter è un componente molto importante dell'architettura CORBA. Uno dei suoi compiti è quello di associare un riferimento a una specifica implementazione nel momento in cui un oggetto è invocato. Quando un client effettua un'invocazione l'adapter collabora con l'ORB e con l'implementazione per fornire il servizio richiesto.

Se un client chiama un oggetto che non è effettivamente in memoria, l'adapter si occupa anche di attivare l'oggetto affinché questo possa rispondere all'invocazione. In molte implementazioni l'adapter può occuparsi anche di disattivare un oggetto non utilizzato da lungo tempo. Dal punto di vista del client l'implementazione è sempre disponibile e caricata in memoria.

Formalmente le specifiche CORBA individuano per l'adapter sei funzionalità chiave:

- Generazione e interpretazione degli object references.
- Invocazione dei metodi attraverso lo skeleton.
- Sicurezza delle interazioni.
- Autenticazione alla chiamata (utilizzando un'entità CORBA detta Principal).
- Attivazione e disattivazione degli oggetti.
- Mapping tra reference e corrispondente implementazione.
- Registrazione dell'implementazione.

Queste funzionalità sono associate al componente logico adapter e nella pratica sono compiute in collaborazione con il core dell'ORB ed eventualmente con altri componenti; alcune funzionalità sono delegate integralmente all'ORB e allo skeleton. L'adapter è comunque coinvolto in ogni invocazione di metodo.

Le funzionalità dell'adapter rendono disponibile via ORB l'implementazione del CORBA object e supportano l'ORB nella gestione del runtime environment dell'oggetto. Dal punto di vista del client l'adapter è il componente che garantisce che le sue richieste siano recapitate a un oggetto attivo in grado di soddisfare la richiesta.

Il meccanismo CORBA opera in modo tale da consentire l'utilizzo contemporaneo di più tipi di adapter con differenti comportamenti (nelle specifiche gli adapter vengono definiti pluggable). A livello di design, l'idea di individuare un'altra entità come l'adapter nasce dalla necessità di modellare in maniera flessibile alcuni aspetti senza estendere l'interfaccia dell'ORB, ma individuando nuovi moduli pluggable.

Il primo tipo di adapter introdotto da OMG è il *Basic Object Adapter* o BOA. Poiché le specifiche del BOA erano lacunose (non definivano ad esempio i meccanismi di attivazione e disattivazione degli oggetti), i vari venditori finirono per realizzarne implementazioni proprietarie largamente incompatibili tra loro che minavano di fatto la portabilità lato server di CORBA. Per questa ragione OMG decise di abbandonare il BOA specificando un nuovo tipo di adapter, il *Portable Object Adapter* o POA.

Nel caso in cui l'ORB scelto supporti il POA sarà sicuramente opportuno utilizzarlo. Esistono tuttavia molti ORB che attualmente non forniscono un'implementazione del POA. Per questa ragione sarà esaminato anche il BOA nonostante il suo utilizzo sia formalmente deprecato. Per gli esempi di questa sezione non sarà possibile utilizzare l'implementazione Sun che non fornisce BOA e POA.

Basic Object Adapter (BOA)

Le specifiche CORBA elencano come compiti primari del BOA la creazione/distruzione degli object reference e il reperimento delle informazioni a questi correlate. Nelle varie implementazioni il BOA, per compiere le sue attività, può accedere al componente proprietario Implementation Repository.

Come si è detto in precedenza, BOA va pensato come un'entità logica e in effetti alcuni dei suoi compiti sono svolti in cooperazione con altri componenti (ad esempio la creazione e la distruzione di object reference sono a carico dello skeleton). Questo ha un impatto sulla sua implementazione che solitamente suddivide i suoi compiti tra il processo ORB, il codice generato dal compilatore IDL e l'effettivo BOA.

Comunque il BOA fornisce l'interfaccia con i metodi necessari a registrare/deregistrare gli oggetti e ad avvertire l'ORB che l'oggetto è effettivamente pronto a rispondere alle invocazioni.

Si è già visto negli esempi precedenti il concetto di server: il server è un'entità eseguibile separata che attiva l'oggetto e gli fornisce un contesto di esecuzione. Anche il BOA per attivare un oggetto si appoggia a un server.

Il server può essere attivato on demand dal BOA (utilizzando informazioni contenute nell'Implementation Repository) oppure da qualche altra entità (ad esempio uno shell script). In ogni caso il server attiverà l'implementazione chiamando il metodo `obj_is_ready` oppure il metodo `impl_is_ready` definiti nel seguente modo

```
// PIDL
```

```
module CORBA {

   interface BOA {

      void impl_is_ready (in ImplementationDef impl);

      void deactivate_impl (in ImplementationDef impl);

      void obj_is_ready (
         in Object obj, in ImplementationDef impl
      );

      void deactivate_obj (in Object obj);

      //...altri metodi di generazione references e access control
   };
};
```

In quasi tutti gli ORB, `obj_is_ready` è il metodo di registrazione del singolo oggetto all'interno di un server e stabilisce un'associazione tra un'istanza e un'entità nell'Implementation Repository.

Il metodo `impl_is_ready` è comunemente implementato come un loop infinito che attende le request del client; il ciclo non cessa fino a quando non viene invocato il `deactivate_impl`.

Esistono molti modi di combinare un server process con l'attivazione di oggetti (un server registra un oggetto, un server registra *n* oggetti, …). Le specifiche CORBA individuano quattro differenti politiche di attivazione.

- Shared Server: il processo server inizializza più oggetti invocando per ognuno `obj_is_ready`. Al termine di queste inizializzazioni il server notifica al BOA, con `impl_is_ready`, la sua disponibilità e rimane attivo fino all'invocazione di `deactivate_impl`. Gli oggetti possono essere singolarmente disattivati con `deactivate_obj`. La disattivazione è quasi sempre automatizzata dal distruttore dello skeleton.

- Unshared Server: ogni oggetto viene associato a un processo server differente. L'inizializzazione avviene comunque con le due chiamate `obj_is_ready` e `impl_is_ready`.

- Server-per-method: un nuovo processo viene creato ad ogni invocazione. Il processo termina al terminare dell'invocazione e, poiché ogni invocazione implica un nuovo processo, non è necessario inviare una notifica al BOA (alcuni ORB richiedono comunque l'invocazione di `impl_is_ready`).

- Persistent Server: tipicamente è un processo avviato mediante qualche meccanismo esterno al BOA (shell script o avvio utente) e va registrato mediante `impl_is_ready`. Dopo la notifica al BOA si comporta esattamente come uno shared server.

A differenza di quanto indicato nelle specifiche, la maggior parte delle implementazioni fornisce un sottoinsieme delle activation policy. In generale l'utilizzo dei metodi legati al BOA è differente tra i vari ORB e genera quasi sempre problemi di portabilità per quanto concerne l'attivazione delle implementazioni.

BOA in pratica

Si provi ora a riscrivere l'applicazione `ShoppingCart` utilizzando l'implementazione BOA fornita da VisiBroker. Per quanto detto sugli adapter si dovrà intervenire sulle classi coinvolte nell'attivazione e nell'invocazione degli oggetti CORBA, andranno quindi modificate le seguenti classi: `ShoppingCartServer`, `ShoppingCartFactoryImpl` e `ShoppingCartClient`.

Si utilizzi il modello di server persistent, una classe Java con metodo main lanciata da linea di comando o da script. La registrazione dell'oggetto `Factory` sarà effettuata via BOA.

Il BOA è un cosiddetto pseudo-object ed è possibile ottenere un riferimento valido ad esso mediante l'invocazione di un metodo dell'ORB: `BOA_init`. Nell'implementazione VisiBroker esistono due differenti metodi `BOA_init` che permettono di ricevere un BOA inizializzato con differenti politiche di gestione dei thread (thread pooling o per session) e di trattamento delle comunicazioni (utilizzo di Secure Socket Layer o no).

Invocando il metodo `BOA_init` senza parametri si otterrà un BOA con le politiche di default (thread pooling senza SSL). Il codice completo della classe server è

```
package server;

import shopping.*;

public class ShoppingCartServer {

   public static void main(String[] args) {

   // Inizializza l'ORB
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

   // Crea l'oggetto Factory
   ShoppingCartFactoryImpl factory
   = new ShoppingCartFactoryImpl("ShoppingCartFactory");

   // Inizializza il BOA
```

```
    // N.B. Utilizzo classi proprietarie
    com.inprise.vbroker.CORBA.BOA boa
    = ((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();

    // Esporta l'oggetto factory
    boa.obj_is_ready(factory);

    System.out.println(factory + " is ready.");

    // Attende le requests
    boa.impl_is_ready();

    }
}
```

Si noti che istanziando l'oggetto `Factory` si è fornita al costruttore la stringa `"ShoppingCartFactory"`. In VisiBroker, specificando un object name quando si istanzia l'oggetto, si ottiene un riferimento di tipo persistent. Il costruttore dell'oggetto dovrà comunque notificare il nome alla superclasse.

```
public ShoppingCartFactoryImpl(String name) {
    super(name);
}
```

L'associazione tra un reference persistent e il nome viene registrata in un tool proprietario denominato OSAgent o ORB Smart Agent.

Pur essendo uno strumento proprietario OSAgent è molto utilizzato nella pratica. Fornisce una versione semplificata di naming service (con VisiBroker viene fornita anche un'implementazione standard del servizio) e implementa alcuni meccanismi proprietari di fault-tolerance e load-balancing. Per una trattazione completa dell'OSAgent si faccia riferimento alla documentazione VisiBroker.

Un riferimento persistent rimane vivo nell'OSAgent anche al termine del processo server (può essere verificato utilizzando il tool `osfind`). Un riferimento transient è invece rigidamente associato al ciclo di vita del server e non può avvalersi dei meccanismi di load-balancing e fault-tolerance forniti dall'OSAgent.

Nell'esempio, l'unico oggetto con riferimento persistent è la `Factory`. I vari carrelli hanno riferimenti transient, sono registrati solo dalla chiamata `obj_is_ready` della `Factory` e sono quindi accessibili solo tramite questa.

Anche utilizzando il BOA, l'oggetto CORBA deve specializzare la classe `<Interfaccia>ImlBase`. Ecco il codice completo della `Factory`.

```
package server;

import shopping.*;
```

```
import java.util.*;

public class ShoppingCartFactoryImpl extends _ShoppingCartFactoryImplBase {

   private Dictionary allCarts = new Hashtable();

   // N.B. Registro nell'OSAgent
   public ShoppingCartFactoryImpl(String name) {
      super(name);
   }

   public synchronized ShoppingCart getShoppingCart(String userID) {

      // Cerca il carrello assegnato allo userID...
      shopping.ShoppingCart cart
      = (shopping.ShoppingCart) allCarts.get(userID);

      // se non lo trova...
      if(cart == null) {

         // crea un nuovo carrello...
         cart = new ShoppingCartImpl();

         // Rende l'oggetto disponibile sull'ORB
         // N.B. _boa() è fornito dalla classe
         // com.inprise.vbroker.CORBA.Object
         _boa().obj_is_ready(cart);

         System.out.println("Created " + userID "'s cart: " + cart);

         // Salva il carrello nel dictionary associandolo allo userID
         allCarts.put(userID, cart);
      }

      // Restituisce il carrello
      return cart;
   }
}
```

Il client può ottenere un riferimento alla `Factory` invocando il metodo `bind` fornito dall'Helper che esegue anche l'opportuno `narrow`. Il codice completo della classe client è:

```
package client;

import shopping.*;
import org.omg.CORBA.*;

public class ShoppingCartClient {
```

```
   public static void main(String args[]) {

      if (args.length != 3) {
         System.err.println("Uso corretto:
                             java ShoppingCartClient userId Autore Titolo");
         return;
      }

      // Crea e inizializza l'ORB
      ORB orb = ORB.init(args, null);

      // Localizza l'oggetto Factory
      ShoppingCartFactory factory
      = ShoppingCartFactoryHelper.bind(orb, "ShoppingCartFactory");

      // Ottengo dalla Factory un oggetto ShoppingCart
      ShoppingCart cart = factory.getShoppingCart(args[0]);

      // Aggiungo un libro
      cart.addBook(new Book(args[1], args[2]));

      // Ottengo la lista dei libri e la stampo
      Book[] list = cart.getBookList();
      for(int i=0; i<list.length; i++)
         System.out.println("Autore " + list[i].Author
                           + " - Titolo " + list[i].Title);

   }
}
```

Prima di avviare il server si attivi l'OSAgent (nel caso si lavori in ambiente distribuito è necessario attivare l'OSAgent sia sulla macchina client che sulla macchina server). Fatto questo, per l'esecuzione dell'esempio si compiano i soliti passi. I riferimenti persistent registrati nell'OSAgent sono controllabili usando il tool `osfind`.

Attivazione automatica con VisiBroker

Si è detto in precedenza che per l'attivazione automatica di un oggetto, l'adapter attiva i necessari processi server utilizzando le informazioni contenute nell'Implementation Repository. In VisiBroker questo meccanismo è fornito dall'*Object Activation Daemon* (OAD).

L'OAD è un repository che mantiene le informazioni sulle classi che un server supporta, sui loro ID e sulle modalità con cui è necessario attivarle. Le informazioni presenti sull'OAD devono essere registrate ed esistono più modalità di registrazione.

Poiché l'OAD è un oggetto CORBA, è possibile costruire un'applicazione che si preoc-

cupi di registrare/deregistrare le varie implementazioni utilizzando i metodi definiti dalla sua interfaccia IDL (per maggior dettagli vedere la documentazione VisiBroker).

Nella pratica è più comune registrare le implementazioni da linea di comando, tipicamente con script di registrazione di più oggetti (magari nella sequenza di boot di una macchina).

Comunque, indipendentemente dalle modalità di registrazione, non si dovrà scrivere codice differente rispetto a quanto fatto in precedenza. Si provi dunque a utilizzare l'attivazione mediante OAD sull'esempio precedentemente scritto per il BOA.

Attivati gli OSAgent si avvii il processo OAD con il comando

```
oad -VBJprop JDKrenameBug
```

con alcune versioni di VisiBroker non sarà necessario utilizzare il flag.

L'OAD va attivato solo sulla macchina su cui si vuole eseguire l'oggetto servant. A questo punto si proceda alla registrazione dell'implementazione sull'OAD.

Il tool da utilizzare è `oadutil` che permette di registrare un'interfaccia CORBA (flag `-i`), con un preciso object name (flag `-o`), indicando il server che sarà utilizzato per l'attivazione (flag `-java`). È anche possibile specificare l'activation policy con il flag `-p`.

Si esegua quindi il comando

```
oadutil reg -i shopping::ShoppingCartFactory -o ShoppingCartFactory
-java server.ShoppingCartServer -p shared
```

L'implementazione sarà a questo punto registrata, ma il server non sarà ancora attivo; è possibile controllare il contenuto dell'OAD con il comando

```
oadutil list –full
```

che fornirà un output del tipo

```
oadutil list: located 1 record(s)

Implementation #1:
-------------------
repository_id        =     IDL:shopping/ShoppingCartFactory:1.0
object_name          =     ShoppingCartFactory
reference data       =
path_name            =     vbj
activation_policy    =     SHARED_SERVER
args                 =     (length=1)[server.ShoppingCartServer; ]
env                  =     NONE

         Nothing active for this implementation
```

Il server sarà attivato alla prima invocazione del client e, poiché si è specificata una politica shared, sarà condiviso anche dai client che successivamente effettueranno una request. Per verificarlo si avvii con le solite modalità l'applicazione client. Lo standard output del processo OAD dovrebbe notificare l'effettiva attivazione del server, in ogni caso è possibile verificare il contenuto dell'OAD con il comando oadutil list visto in precedenza.

In un contesto reale l'attivazione con queste modalità semplifica decisamente le attività di gestione e manutenzione risultando preferibile rispetto all'attivazione manuale dei server.

Nel caso in cui si debbano attivare molti oggetti, può essere necessario attivare/disattivare in maniera mirata i vari servant; un meccanismo del genere è realizzabile fornendo un cosiddetto service activator.

In linea generale un service activator raggruppa *n* oggetti per i quali è in grado di determinare l'attivazione/disattivazione ad ogni request. Per definire le operazioni di activate/deactivate, l'Activator dovrà implementare l'interfaccia `com.visigenic.vbroker.extension.Activator`. Per una trattazione dell'Activator si rimanda alla documentazione VisiBroker.

Per l'utilizzo di OAD valgono le stesse considerazioni viste in precedenza per OSAgent: fornisce un più semplice utilizzo e notevoli possibilità, ma limita la portabilità lato server. Per una completa portabilità lato server è opportuno utilizzare POA.

Portable Object Adapter (POA)

Il POA entra a far parte delle specifiche CORBA nel 1997 e va a sostituire integralmente a livello funzionale le precedenti specifiche BOA.

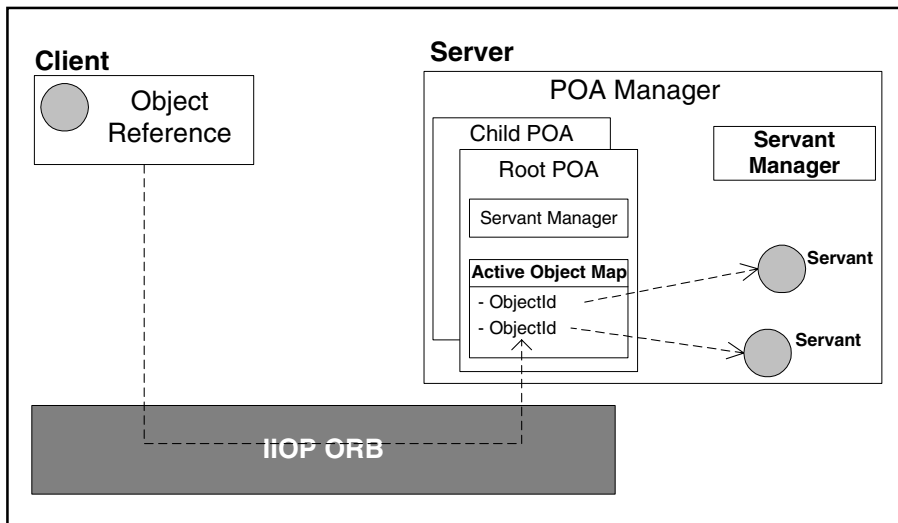
La scelta di sostituire integralmente le API BOA è legata all'impossibilità di coniugare i complessi sviluppi presi dalle varie implementazioni proprietarie. Poiché l'ORB è pensato per supportare un numero arbitrario di adapter, BOA e POA possono comunque coesistere.

Lo sviluppo del POA prende l'avvio e si basa sulle molteplici esperienze derivate dalle varie implementazioni del BOA (spesso si dice che POA è semplicemente una versione corretta di BOA), è quindi chiaro che molteplici sono i concetti comuni alle due tipologie di adapter.

Anche per POA valgono dunque le distinzioni effettuate sull'attivazione di implementazioni, sulle tipologie di server e la distinzione tra oggetti persistent o transient.

Per ogni implementazione è definibile un servant manager che, invocato dal POA, crea, attiva o disattiva i vari servant on demand. Il meccanismo dei servant manager aiuta il POA nella gestione degli oggetti server-side. Quasi sempre gli ORB forniscono dei servant manager di default che implementano politiche definite.

È comunque possibile definire direttamente il set di politiche applicate al server senza utilizzare un servant manager, ma operando direttamente sul POA. Nel caso in cui si

Figura 16.8 – *Funzionamento POA*

utilizzi un servant manager, sarà suo il compito di associare la request a un preciso servant, attivandolo o creandolo se necessario.

Un servant manager implementa una delle due interfacce di callback `ServantActivator` e `ServantLocator`. In generale l'`Activator` si riferisce a oggetti di tipo persistent, mentre il `Locator` si riferisce a oggetti di tipo transient.

Indipendentemente da quale interfaccia si utilizzi, le operazioni da definire sono due, una per reperire e restituire il servant, l'altra per disattivarlo. Nel caso di `ServantActivator` le due operazioni di cui sopra sono `incarnate` ed `etherealize`, mentre nel caso di `ServantLocator` sono `preinvoke` e `postinvoke`.

Il POA mantiene una mappa (Active Object Map) dei servant attivi in ogni istante. All'interno della mappa i servant sono associati a uno o più Object Id. Un riferimento a un oggetto sul lato client incapsula l'Object Id e il riferimento al POA ed è utilizzato sul lato server da ORB, POA e servant manager per recapitare la request a un preciso servant.

Non esiste una forma standard dell'Object Id che può essere generato dal POA oppure essere assegnato dall'implementazione. In ogni caso l'Object Id deve essere unico nel namespace e quindi nel POA su cui è mantenuto.

La struttura dei POA è ad albero a partire da un RootPOA. Il RootPOA è sempre disponibile e possiede politiche di default. A partire da questo è possibile generare una gerarchia di nodi POA child con politiche differenti. Sul RootPOA sono mantenuti solo riferimenti transient ed è per questo che nella pratica gli oggetti si installano quasi sempre su child POA creati opportunamente.

Un Child POA può essere creato invocando il metodo `create_POA` sul suo nodo padre. Per definire le politiche del POA creato bisogna invocare il metodo di creazione fornendogli un oggetto di tipo `Policy` opportunamente inizializzato. Non è possibile modificare successivamente le politiche di un nodo POA.

Ecco la definizione dei metodi che gestiscono il ciclo di vita di un POA.

```
// IDL
module PortableServer {

   //...

   interface POA {

      //...

      // POA creation and destruction

      POA create_POA(in string adapter_name, in POAManager a_POAManager,
                     in CORBA::PolicyList policies) raises (AdapterAlreadyExists,
                                                            InvalidPolicy);

      POA find_POA(in string adapter_name,
                   in boolean activate_it) raises (AdapterNonExistent);

      void destroy(in boolean etherealize_objects,
                   in boolean wait_for_completion);
   };
};
```

L'oggetto `Policy` va a coprire vari aspetti del runtime environment di un servant associato a un preciso POA. Le specifiche CORBA individuano sette differenti aspetti (in corsivo sono indicati i default):

- Thread: specifica le modalità di trattamento dei thread ovvero singolo thread (`SINGLE_THREAD_MODEL`) o multithread (*`ORB_CTRL_MODEL`*).
- Lifespan: specifica il modello di persistenza (`PERSISTENT` o *`TRANSIENT`*).
- Object Id Uniqueness: specifica se l'Id di un servant deve essere unico (*`UNIQUE_ID`*) o può essere multiplo (`MULTIPLE_ID`).
- Id Assignment: specifica se l'Id deve essere assegnato dall'applicazione (`USER_ID`) o dal POA (*`SYSTEM_ID`*).

- Servant Retention: specifica se il POA mantiene i servant nell'Active Object Map (*RETAIN*) o si affida a un servant manager (NON_RETAIN).

- Activation: specifica se il POA supporta l'attivazione implicita dei servant (*IMPLICIT_ACTIVATION* o NO_ IMPLICIT_ACTIVATION).

- Request Processing: specifica come vengono processate le request; usando l'Active Object Map (*USE_ACTIVE_OBJECT_MAP_ONLY*, da usare con RETAIN) o un servant manager (USE_DEFAULT_SERVANT o USE_SERVANT_MANAGER, da usare con NON_RETAIN).

Per ognuna di queste categorie il POA offre una Factory del tipo create_XXX_policy (create_thread_policy, create_lifespan_policy, …); la costruzione di un oggetto di tipo Policy avviene utilizzando la Factory opportuna.

La scelta della combinazione da adottare è legata alla tipologia di applicazione che si sta realizzando. Ad esempio, una combinazione di RETAIN e USE_ACTIVE_OBJECT_MAP_ONLY (il default) può essere accettabile per applicazioni che gestiscono un numero finito di oggetti attivati all'avvio (come un Application Server che fornisca servizi continuativamente), ma è una soluzione troppo limitata per un'applicazione che gestisca un gran numero di oggetti. Per applicazioni di questo tipo sono più opportune soluzioni RETAIN che utilizzino servant manager. USE_SERVANT_MANAGER è più indicato per un gran numero di oggetti persistent, mentre USE_DEFAULT_SERVANT per un gran numero di oggetti transient.

Esistono tre differenti tipologie di attivazione di un oggetto mediante POA: attivazione esplicita, attivazione on demand utilizzando un servant manager, e attivazione implicita. L'attivazione esplicita avviene con l'invocazione di alcuni metodi forniti dal POA.

```
// IDL
module PortableServer {

   //...

   interface POA {

      //...

      // object activation and deactivation

      ObjectId activate_object(in Servant p_servant) raises (ServantAlreadyActive,
                                                             WrongPolicy);

      void activate_object_with_id(in ObjectId id,
                              in Servant p_servant) raises (ServantAlreadyActive,
                                                             ObjectAlreadyActive,
                                                             WrongPolicy);
```

```
      void deactivate_object(in ObjectId oid) raises (ObjectNotActive,
                                                       WrongPolicy);

   };
};
```

Nel caso in cui si usi la registrazione esplicita, il server crea tutti i servant e li registra con uno dei due metodi `activate_object`.

Nel caso di attivazione on demand, il server si limita a informare il POA su quale servant manager utilizzare per l'attivazione degli oggetti. I metodi per la gestione dei servant manager sono

```
// IDL
module PortableServer {

   //...

   interface POA {

      //...

      // Servant Manager registration

      ServantManager get_servant_manager() raises (WrongPolicy);

      void set_servant_manager(in ServantManager imgr) raises (WrongPolicy);

   };
};
```

Si ha un'attivazione implicita effettuando su di un servant inattivo, senza Object Id, operazioni che implichino la presenza di un Object Id nell'Active Map. È possibile solo per la combinazione `IMPLICIT_ACTIVATION`, `SYSTEM_ID`, `RETAIN`. Le operazioni che generano un'attivazione implicita sono

```
// IDL
module PortableServer {

   //...

   interface POA {

      //...

      // Identity mapping operations
```

```
      ObjectId servant_to_id(in Servant p_servant) raises (ServantNotActive,
                                                           WrongPolicy);

      Object servant_to_reference(in Servant p_servant) raises (ServantNotActive,
                                                                WrongPolicy);

   };
};
```

Anche il POA può essere attivato e disattivato; queste operazioni possono essere effettuate utilizzando il `POAManager` che è un oggetto associato a uno o più POA. Il `POAManager` permette anche di bloccare e scartare le request in arrivo.

```
// IDL
module PortableServer {

   //...

   // POAManager interface
   interface POAManager {

      exception AdapterInactive {};

      enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

      void activate() raises(AdapterInactive);

      void hold_requests(in boolean wait_for_completion) raises(AdapterInactive);

      void discard_requests(in boolean wait_for_completion) raises(AdapterInactive);

      void deactivate(in boolean etherealize_objects,
                      in boolean wait_for_completion) raises(AdapterInactive);

      State get_state();
   };
};
```

POA in pratica

L'utilizzo di POA permette di configurare l'environment e il comportamento del servant in maniera indipendente dall'ORB. Sarà presentato il solito esempio dello Shopping Cart avendo come punto di riferimento VisiBroker. L'esempio sarà comunque utilizzabile con qualunque altro ORB dotato di POA.

Il modo più semplice di utilizzare POA è quello di adottare l'attivazione esplicita senza definire servant manager. Sarà quindi realizzato un server che attivi il servant

`ShoppingCartFactory` in modo persistent. La Factory sarà un servizio sempre disponibile, mentre i singoli carrelli, come già nella versione BOA, saranno oggetti transient. Il primo passo da compiere è quello di ottenere un riferimento al `RootPOA`.

```
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

Fatto questo è possibile creare un POA con le necessarie politiche. Non è possibile utilizzare direttamente il `RootPOA` in quanto non supporta la modalità persistent.

Un POA di default utilizza modalità multithread, riferimenti transient, Active Object Map. Nell'esempio è quindi sufficiente modificare la proprietà Lifespan da `TRANSIENT` a `PERSISTENT` e, con queste politiche, creare il nuovo nodo POA assegnandogli un nome identificativo.

```
org.omg.CORBA.Policy[] policies = {
   rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};

POA myPOA = rootPOA.create_POA("shopping_cart_poa",
                              rootPOA.the_POAManager(), policies);
```

A questo punto è possibile istanziare il servant e attivarlo sul POA. Poiché l'Id deve essere conosciuto dal client per l'invocazione, in uno scenario semplice è conveniente definirlo esplicitamente nel server.

```
byte[] factoryId = "ShoppingCartFactory".getBytes();

myPOA.activate_object_with_id(factoryId, factory);
```

Il servant non sarà in condizione di rispondere fino all'attivazione del POA che lo ospita. L'attivazione è effettuata utilizzando il POAManager.

```
rootPOA.the_POAManager().activate();
```

Il ciclo di attesa infinita del server può essere implementato con un metodo dell'ORB (non più `impl_is_ready`).

```
orb.run();
```

Ecco il codice completo della classe server.

```
package server;

import shopping.*;
```

```
import org.omg.PortableServer.*;

public class ShoppingCartServer {

   public static void main(String[] args) {

   try {

      // Inizializza l'ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      // Prende il reference al the root POA
      POA rootPOA
      = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      // Crea le policies per il persistent POA
      org.omg.CORBA.Policy[] policies = {
         rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };

      // Crea myPOA con le date policies
      POA myPOA = rootPOA.create_POA("shopping_cart_poa",
                              rootPOA.the_POAManager(), policies);

      // Crea l'oggetto Factory
      ShoppingCartFactoryImpl factory = new ShoppingCartFactoryImpl();

      // Stabilsco l'ID del servant
      byte[] factoryId = "ShoppingCartFactory".getBytes();

      // Attiva il servant su myPOA con il dato Id
      myPOA.activate_object_with_id(factoryId, factory);

      // Attiva il POA
      rootPOA.the_POAManager().activate();

      System.out.println(myPOA.servant_to_reference(factory) + " is ready.");

      // Si mette in attesa delle requests
      orb.run();

   }
   catch (Exception e) {
      e.printStackTrace();
   }
   }
}
```

Utilizzando POA un servant non deve più specializzare `_<NomeInter-`

`faccia>ImplBase`, bensì `<NomeInterfaccia>POA`. La `ShoppingCartFactoryImpl` sarà quindi

```
package server;

import org.omg.PortableServer.*;
import shopping.*;
import java.util.*;

public class ShoppingCartFactoryImpl extends ShoppingCartFactoryPOA {

   private Dictionary allCarts = new Hashtable();

   public synchronized ShoppingCart getShoppingCart(String userID) {

   // Cerca il carrello assegnato allo userID...
   shopping.ShoppingCart cart = (shopping.ShoppingCart) allCarts.get(userID);

   // se non lo trova...
   if(cart == null) {

      // crea un nuovo carrello...
      ShoppingCartImpl cartServant = new ShoppingCartImpl();

      try {

         // Attiva l'oggetto sul default POA che
         // è il root POA di questo servant
         cart = shopping.ShoppingCartHelper.narrow(
                            _default_POA().servant_to_reference(cartServant));

      } catch (Exception e) {
      e.printStackTrace();
      }

      System.out.println("Created " + userID + "'s cart: " + cart);

      // Salva il carrello associandolo allo userID
      allCarts.put(userID, cart);
      }

      // Restituisce il carrello
      return cart;
   }
}
```

Si noti che l'attivazione del singolo `ShoppingCart` è implicita. La generazione dell'Id e la sua registrazione nell'Active Objec Map saranno in questo caso a carico del POA.

L'oggetto `ShoppingCartImpl` dovrà specializzare `ShoppingCartPOA`; il codice rimarrà inalterato rispetto all'esempio BOA.

Il client accederà alla Factory in modo molto simile a quanto visto nel BOA. Sul bind andrà specificato l'opportuno Id e il nome del POA su cui è registrato il servant.

```
byte[] factoryId = "ShoppingCartFactory".getBytes();

ShoppingCartFactory factory
= ShoppingCartFactoryHelper.bind (orb, "/shopping_cart_poa", factoryId);
```

Ecco il codice completo della classe client.

```
package client;

import shopping.*;
import org.omg.CORBA.*;

public class ShoppingCartClient {

   public static void main(String args[]) {

      if (args.length != 3) {
         System.err.println("Uso corretto:
                             java ShoppingCartClient userId Autore Titolo");
         return;
      }

      // Crea e inizializza l'ORB
      ORB orb = ORB.init(args, null);

      // ID del servant
      byte[] factoryId = "ShoppingCartFactory".getBytes();

      // Localizza l'oggetto Factory
      // Devo usare il POA name e l'Id del servant
      ShoppingCartFactory factory
      = ShoppingCartFactoryHelper.bind(orb, "/shopping_cart_poa", factoryId);

      // Ottengo dalla Factory un oggetto ShoppingCart
      ShoppingCart cart = factory.getShoppingCart(args[0]);

      // Aggiungo un libro
      cart.addBook(new Book(args[1], args[2]));

      // Ottengo la lista dei libri e la stampo
      Book[] list = cart.getBookList();
      for(int i=0; i<list.length; i++)
      System.out.println("Autore " + list[i].Author
                         + " - Titolo " + list[i].Title);
```

```
    }
}
```

Parametri per valore

Nel progettare applicazioni distribuite, tipicamente si ragiona suddividendo il dominio dell'applicazione in layer. Il modello più celebre individua tre livelli: presentation, business logic e data/resources. I differenti livelli comunicano attraverso un canale, ma non devono condividere alcun oggetto a livello d'implementazione. Anche per questa ragione middleware come RMI o CORBA operano una netta separazione tra interfaccia e implementazione.

A titolo d'esempio, si immagini un client che si occupa solamente di presentation e un server (magari su un'altra macchina) che gestisce la business logic. Le due applicazioni dovranno al più condividere oggetti che incapsulino dati, ma non dovranno condividere alcun metodo (un metodo di presentation è necessario sul client, un metodo di business è necessario sul server). Per questi scopi sono quindi sufficienti le strutture dati fornite da IDL (`struct`, `enum`, …).

Nonostante queste considerazioni, è spesso utile ricevere/restituire oggetti per valore. In molti casi può essere comodo avere metodi utilizzabili localmente al client e al server.

Fino alle specifiche 2.3 non era possibile il passaggio per valore con oggetti CORBA; come si è visto finora le interfacce sono sempre trattate per riferimento. A differenza di RMI, i metodi non potevano quindi restituire/ricevere via serializzazione oggetti condivisi da client e server. L'introduzione del concetto di valuetype ha ovviato a questa mancanza.

Data la recente introduzione del `valuetype`, alcuni ORB, tra cui quello del JDK 1.2, non supportano ancora questa specifica. Per questa ragione nelle sezioni successive si studieranno il `valuetype` e alcuni approcci alternativi.

Una possibile soluzione

La prima soluzione è molto semplice e in realtà non è un vero passaggio per copia. L'idea è quella di avere una classe locale (sul server o sul client) che fasci la struttura definita nell'IDL. Questa classe dovrà avere i metodi da utilizzare localmente e, per comodità, un costruttore che riceva in input la struttura dati.

Si provi quindi a implementare una semplice (quanto fasulla) funzionalità di login.

```
// IDL
module authentication {

   struct User {
      string userId;
   };
```

```
   exception InvalidUserException{};

   interface UserLogin {
      User login(in string user, in string pwd) raises (InvalidUserException);
   };

};
```

Per semplicità si supponga di impostare, sul client e sul server, userId e password da linea di comando (in un contesto reale i dati utente risiederebbero su repository quali basi dati, files o directory server). Il server provvederà a impostare i valori di userId e password sul servant

```
UserLoginImpl login = new UserLoginImpl(args[0], args[1]);
```

Il server effettua le stesse procedure di attivazione e registrazione via Naming Service viste in precedenza. Il codice completo della classe server non viene mostrato.

Ecco invece il codice completo della classe servant.

```
package server;

import authentication.*;

public class UserLoginImpl extends _UserLoginImplBase {

   private String userId;
   private String pwd;

   public UserLoginImpl(String u, String p) {
      super();

      userId = u;
      pwd = p;
   }

   // Metodo di Login
   public User login(String u, String p) throws InvalidUserException {

      if (userId.equals(u) && pwd.equals(p))
         return new User(u);
      else
         throw new InvalidUserException();

   }
}
```

È possibile ora definire il wrapper dello User client-side.

```
package client;

import authentication.*;

public class UserLocalWithMethods {

   private User user;

   public UserLocalWithMethods(User user) {
      this.user = user;
   }

   // Metodo della classe locale che accede alla struct User
   public String getUserId() {
      return user.userId;
   }

   // Override del metodo toString
   public String toString() {
      return "#User : " + user.userId;
   }
}
```

L'oggetto wrapper sarà creato incapsulando la classe `User` (che rappresenta la struct IDL). Sull'oggetto sarà possibile invocare i metodi definiti localmente (nel caso in esame `getUserId` e `toString`).

```
UserLogin uLogin = UserLoginHelper.narrow(ncRef.resolve(path));

// Effettuo il login e creo l'oggetto wrapper
UserLocalWithMethods user
= new UserLocalWithMethods(uLogin.login(args[0], args[1]));

// Utilizzo il metodo della classe locale
System.out.println("Login UserId: " + user.getUserId());

// Utilizzo il metodo toString della classe locale
System.out.println(user);
```

Il client e il server non condivideranno l'implementazione di alcun metodo, ma si limiteranno a condividere la rappresentazione della struttura IDL. La classe con i metodi andrà distribuita semplicemente sul layer logicamente destinato alla sua esecuzione. Potranno esistere wrapper differenti per layer differenti.

Questa soluzione, pur non essendo un effettivo passaggio per valore, rappresenta l'implementazione formalmente più corretta e più vicina allo spirito originale CORBA.

Serializzazione in CORBA

La seconda soluzione utilizza la serializzazione Java e quindi, non essendo portabile, non è molto in linea con lo spirito CORBA. Nel caso però in cui si affronti uno scenario che prevede Java sui client e sui server è comunque una soluzione comoda, simile nell'approccio a RMI. Si ridefinisca l'IDL vista in precedenza

```
module authentication {

   typedef sequence <octet> User;

   exception InvalidUserException{};

   interface UserLogin {
      User login(in string user, in string pwd) raises (InvalidUserException);
   };

};
```

In questo modo il tipo `User`, definito come sequence di octet, sarà effettivamente tradotto in Java come array di byte. Il metodo `login` potrà quindi restituire qualunque oggetto serializzato. L'oggetto condiviso da client e server dovrà essere serializzabile

```
package client;

import java.io.*;

public class User implements Serializable {

   private String userId;

   public User(String userId) {
      this.userId = userId;
   }

   public String getUserId() {
      return userId;
   }

   // Override del metodo toString
   public String toString() {
      return "#User : " + userId;
   }
}
```

L'effettiva serializzazione sarà operata dal metodo login del servant modificato come di seguito riportato.

```
public byte[] login(String u, String p) throws InvalidUserException {

   if (userId.equals(u) && pwd.equals(p)) {

      // Serializza un oggetto user in un array di byte
      byte[] b = serializza(new client.User(u));
      return b;

   } else
      throw new InvalidUserException();

}
```

Il metodo utilizzato per ottenere l'array di byte serializza un generico oggetto

```
public byte[] serializza(java.lang.Object obj) {

   ByteArrayOutputStream bOut = null;

   try {

      bOut = new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bOut);
      out.writeObject(obj);

   } catch(Exception e) {
      e.printStackTrace();
   }

   return bOut.toByteArray();
}
```

Il client opererà in maniera speculare

```
UserLogin uLogin = UserLoginHelper.narrow(ncRef.resolve(path));

// Effettuo il login
byte[] b = uLogin.login(userId, pwd);

// Ottengo l'oggetto User serializzato
User user = (User) deserializza(b);

// Utilizzo il metodo della classe serializzata
System.out.println("Login UserId: " + user.getUserId());

// Utilizzo il metodo toString della classe serializzata
System.out.println(user);
```

Il metodo `deserializza` è definito come segue:

```
public java.lang.Object deserializza(byte[] b) {

   java.lang.Object obj = null;

   try {

      ByteArrayInputStream bIn = new ByteArrayInputStream(b);
      ObjectInputStream oIn = new ObjectInputStream(bIn);
      obj = oIn.readObject();

   } catch(Exception e) {
      e.printStackTrace();
   }

   return obj;
}
```

Questa soluzione mina l'interoperabiltà con altri linguaggi e inoltre, dal momento che tratta i parametri come array di byte e non come tipi, diminuisce il livello espressivo di un'interfaccia.

Valuetype

Le soluzioni precedenti sono semplici approcci applicativi; le specifiche CORBA 2.3 hanno definito un approccio standard al passaggio di oggetti per valore. Questa parte di specifiche riveste una grandissima importanza in quanto è uno degli elementi chiave di avvicinamento tra RMI e CORBA. È uno dei meccanismi fondamentali per i tool che implementano RMI over IIOP (si veda più avanti, per una discussione più approfondita).

L'idea chiave che sta alla base delle specifiche CORBA *Object-by-Value* (OBV) è quella di fornire una sorta di serializzazione multipiattaforma.

La definizione di un oggetto serializzabile può essere suddivisa in stato e implementazione. La componente stato è sostanzialmente riconducibile ai valori che hanno gli attributi ed è quindi legata alla singola istanza (escludendo ovviamente attributi `static`), la seconda componente è l'implementazione dei metodi ed è comune a tutte le istanze.

Anche in Java la serializzazione si limita a rendere persistente lo stato. In fase di deserializzazione l'oggetto viene ricostruito utilizzando la sua definizione (la classe) a partire dal suo stato.

Per la natura delle specifiche CORBA, la definizione di un meccanismo simile a quello descritto comporta un'estensione del linguaggio IDL. La keyword `valuetype` consente di specificare un nuovo tipo che utilizza il passaggio per valore.

Si modifichi l'IDL vista in precedenza definendo l'oggetto `User` come `valuetype`.

```
// IDL
module authentication {

   valuetype User {
      // Metodi locali
      string getUserId();

      // Stato
      private string userId;
   };

   exception InvalidUserException{};

   interface UserLogin {
      User login(in string user, in string pwd) raises (InvalidUserException);
   };

};
```

La definizione mediante `valuetype` consente di specificare gli attributi con gli opportuni modificatori di accesso (`private` e `public`) e le signature dei metodi definite con le stesse modalità adottate nelle interface.

Compilando l'IDL si ottiene la seguente definizione del tipo `User` (l'esempio non è utilizzabile con il JDK 1.2).

```
package authentication;

public abstract class User implements org.omg.CORBA.portable.StreamableValue {

   protected java.lang.String userId;

   abstract public java.lang.String getUserId ();

   //...

}
```

È necessario fornire una versione concreta dell'oggetto `User` a partire dalla classe astratta ottenuta dalla precompilazione. Si definisca allora `UserImpl` come segue:

```
package authentication;

public class UserImpl extends User {

   public UserImpl() {
   }
```

```
   public UserImpl(String userId) {
      this.userId = userId;
   }

   public String getUserId() {
      return userId;
   }

   public String toString() {
      return "#User : " + userId;
   }
}
```

Quando l'ORB riceve un valuetype deve effettuare l'unmarshalling e quindi creare una nuova istanza dell'oggetto opportunamente valorizzata; per fare questo utilizza la `Factory` associata al tipo in questione. Una factory di default viene creata per ogni valuetype dal processo di compilazione dell'IDL. Nel caso in esame sarà generata una classe `UserDefaultFactory`.

La classe generata può essere utilizzata come base per la definizione di `Factory` complesse o semplicemente modificata per ottenere il comportamento voluto; in ogni caso l'ORB deve conoscere l'associazione valuetype–factory. L'associazione può essere stabilita esplicitamente utilizzando il metodo `register_value_factory` dell'ORB oppure implicitamente utilizzando le naming convention che stabiliscono che, nel caso in cui non esista un'associazione esplicita, l'ORB utilizzi la classe `<valuetype>DefaultFactory`.

Per semplicità si adotti il secondo approccio. Nel caso si utilizzi idlj di Sun la `UserDefaultFactory` generata non necessita di modifiche. Invece, nel caso si utilizzi VisiBroker, la classe generata dall'IDL sarà incompleta e dovrebbe presentare il codice seguente

```
package authentication;

public class UserDefaultFactory implements org.omg.CORBA.portable.ValueFactory {


public java.io.Serializable read_value(org.omg.CORBA_2_3.portable.InputStream is) {

      // INSTANTIATE IMPLEMENTATION CLASS ON THE LINE BELOW
      java.io.Serializable val = null;

      // REMOVE THE LINE BELOW AFTER FINISHING IMPLEMENTATION
      throw new org.omg.CORBA.NO_IMPLEMENT();
      return is.read_value(val);
   }
}
```

Con una versione di JDK differente dalla 1.2 il codice generato potrebbe essere diverso e presentare problemi di compilazione.

I commenti generati da idl2java indicano quali modifiche effettuare. Per il caso in esame la `Factory` può limitarsi a istanziare `UserImpl`.

```
package authentication;

public class UserDefaultFactory implements org.omg.CORBA.portable.ValueFactory {

   public java.io.Serializable read_value(
                                   org.omg.CORBA_2_3.portable.InputStream is) {
      java.io.Serializable val = new UserImpl();
      return is.read_value(val);
   }
}
```

La classe `Factory` dovrà comunque essere presente nell'ambiente destinato all'unmarshalling (nell'esempio sul client). La restituzione dell'oggetto sarà effettuata dal metodo `login` di `UserLoginImpl` modificato come segue

```
public User login(String u, String p) throws InvalidUserException {

   if (userId.equals(u) && pwd.equals(p))
      return new UserImpl(u);
   else
      throw new InvalidUserException();

}
```

Dal punto di vista del client il meccanismo valuetype è invece assolutamente trasparente.

```
User user = uLogin.login(args[0], args[1]);

System.out.println("Login UserId: " + user.getUserId());
```

Nel caso in cui l'ORB non riesca a individuare un'implementazione per il tipo ricevuto, potrà provare a scaricare la corretta implementazione dal chiamante (la funzionalità è simile a quella del codebase RMI). Questa funzionalità può essere disabilitata per ragioni di security.

Come già detto, l'implementazione del meccanismo object-by-value in CORBA ha grande importanza perché consente un semplice utilizzo di RMI over IIOP. È quindi significativo per le specifiche EJB 1.1 che hanno indicato come standard l'adozione di RMI over IIOP. Limitandosi invece allo scenario programmazione distribuita CORBA, valuetype è importante poiché introduce una sorta di serializzazione multilinguaggio e la semantica null value.

CORBA runtime information

In Java esiste la possibilità di ottenere a runtime una serie d'informazioni sulla composizione di un qualunque oggetto (costruttori, metodi, attributi, ecc.). Grazie a queste informazioni è possibile utilizzare un'istanza (invocare metodi, impostare attributi, ecc.) pur non conoscendo a priori la sua classe. Questa potenzialità consente di realizzare soluzioni molto eleganti e flessibili a problemi storicamente complessi quali la definizione/manipolazione di componenti.

CORBA fornisce meccanismi simili a quelli appena descritti e tutti i CORBA object possono fornire a runtime informazioni sulla propria struttura. Nel gergo CORBA queste informazioni sono dette metadata e pervadono l'intero sistema distribuito. Il repository destinato a contenere i metadata è l'Interface Repository che consente di reperire e utilizzare un oggetto ottenendo dinamicamente la sua descrizione IDL.

Queste caratteristiche conferiscono una flessibilità notevole al modello CORBA. Nella pratica risultano particolarmente attraenti per i venditori di tool che sviluppano componenti distribuiti. L'invocazione dinamica, presentando minori performance e maggiori difficoltà di sviluppo, è sicuramente meno interessante per un utilizzo medio.

Introspezione CORBA

Ogni oggetto CORBA specializza `CORBA::Object`; le capacità introspettive di ogni CORBA object derivano da questa superclasse.

```
// IDL
module CORBA {
   //...
   interface Object {
      //...
      IRObject get_interface_def();

      ImplementationDef get_implementation();

      boolean is_nil();

      boolean is_a(in string logical_type_id);

      boolean is_equivalent(in Object other_object);

      boolean non_existent();
   };
};
```

Il metodo `get_interface_def` può essere visto come l'equivalente del `getClass Java` e fornisce i dettagli relativi a un oggetto (attributi, metodi, ecc.). Il metodo restitu-

isce un oggetto generico di tipo `IRObject`, per ottenere i dettagli di cui sopra è necessario effettuare il narrow a `InterfaceDef`. Fino alle specifiche 2.3 esisteva un metodo `get_interface` che restituiva direttamente un oggetto di tipo `InterfaceDef`: molti ORB lo supportano ancora per backward compatibility.

I restanti metodi sono funzioni di test: `is_nil` è l'equivalente CORBA di `"obj == null"`, `is_a` controlla la compatibilità dell'oggetto con l'interfaccia data, `is_equivalent` verifica l'equivalenza tra due reference e `non_existent` verifica se l'oggetto non è più valido.

Va notato che le signature viste sopra rappresentano le signature IDL del `CORBA::Object`, ma nella pratica tutti i metodi visti sopra sono forniti dal `CORBA.Object` con un " _ " davanti (`_is_a`, `_get_implementation`, …).

Interface Repository (IR)

L'Interface Repository è il servizio che contiene e fornisce le informazioni sulla struttura IDL degli oggetti. Si è visto nei paragrafi precedenti che, attraverso l'invocazione del metodo `get_interface_def`, è possibile ottenere un oggetto di tipo `InterfaceDef` che fornisca tutte le informazioni sulla struttura di una interfaccia.

L'IR contiene in forma persistente gli oggetti `InterfaceDef` che rappresentano le interfacce IDL registrate presso il repository. L'IR può esser utilizzato anche dall'ORB per effettuare il type-checking nell'invocazione dei metodi, per assistere l'interazione tra differenti implementazioni di ORB o per garantire la correttezza del grafo di derivazione.

Nel caso in cui si intenda utilizzare esplicitamente l'IR sarà necessario registrare l'IDL. La registrazione varia nelle varie implementazioni. VisiBroker prevede due comandi: `irep` (attiva l'IR) e `idl2ir` (registra un IDL presso l'IR). Il JDK non fornisce attualmente alcuna implementazione dell'IR.

Interrogando l'IR è possibile ottenere tutte le informazioni descrivibili con un'IDL. La navigazione di queste informazioni avviene tipicamente a partire dalla classe `InterfaceDef` e coinvolge un complesso insieme di classi che rispecchiano l'intera specifica IDL: `ModuleDef`, `InterfaceDef`, `OperationDef`, `ParameterDef`, `AttributeDef`, `ConstantDef`, …

Dynamic Invocation Interface (DII)

Negli esempi visti finora, per invocare i metodi sugli oggetti CORBA si è utilizzata l'invocazione statica. Questo modello di invocazione richiede la precisa conoscenza, garantita dalla presenza dello stub, dell'interfaccia dell'oggetto.

In realtà in CORBA è possibile accedere a un oggetto, scoprire i suoi metodi ed eventualmente invocarli, senza avere alcuno stub precompilato e senza conoscere a priori l'interfaccia esposta dall'oggetto CORBA.

Questo implica la possibilità di scoprire le informazioni di interfaccia a runtime. Ciò è possibile utilizzando una delle più note caratteristiche CORBA: la *Dynamic Invocation Interface* (DII). La DII opera soltanto nei confronti del client; esiste un meccanismo analogo lato server (*Dynamic Skeleton Interface*, DSI) che consente a un ORB di dialogare con un'implementazione senza alcuno skeleton precompilato.

Purtroppo, anche se DII fa parte del core CORBA, le sue funzionalità sono disperse su un gran numero di oggetti. Per invocare dinamicamente un metodo su di un oggetto i passi da compiere sono:

- ottenere un riferimento all'oggetto: anche con DII per utilizzare un oggetto è necessario ottenere un riferimento valido. Il client otterrà un riferimento generico ma, non avendo classi precompilate, non potrà effettuare un `narrow` o utilizzare meccanismi quali il bind fornito dall'Helper.

- ottenere l'interfaccia: invocando il metodo `get_interface_def` sul riferimento si ottiene un `IRObject` e da questo, con narrow, l'oggetto navigabile di tipo `InterfaceDef`. Questo oggetto è contenuto nell'IR e consente di ottenere tutte le informazioni di interfaccia. Con i metodi `lookup_name` e `describe` di `InterfaceDef` è possibile reperire un metodo e una sua completa descrizione.

- creare la lista di parametri: per definire la lista di parametri viene usata una struttura dati particolare, `NVList` (Named Value List). La creazione di una `NVList` è effettuata o da un metodo dell'ORB (`create_operation_list`) o da un metodo di Request (`arguments`). In ogni caso con il metodo `add_item` è possibile comporre la lista di parametri.

- creare la Request: la Request incapsula tutte le informazioni necessarie all'invocazione di un metodo (nome metodo, lista argomenti e valore di ritorno). Comporre la Request è la parte più pesante e laboriosa della DII. Può essere creata invocando sul reference dell'oggetto il metodo `create_request` o la versione semplificata `_request`.

- invocare il metodo: utilizzando Request esistono più modi di invocare il metodo. Il primo modo è quello di invocarlo in modalità sincrona con `invoke`. Il secondo modo è quello di invocarlo in modalità asincrona con `send_deferred` e controllare in un secondo momento la risposta con `poll_response` o `get_response`. Il terzo modo è l'invocazione senza response con `send_oneway`.

Come si può osservare, un'invocazione dinamica può essere effettuata seguendo percorsi differenti. Lo scenario più complesso implica un'interazione con l'Interface Repository

(il secondo e il terzo passo dell'elenco precedente) mediante la quale è possibile ottenere l'intera descrizione di un metodo.

Scenari più semplici non prevedono l'interazione con l'IR e sono a esempio adottati quando ci si limita a pilotare l'invocazione dinamica con parametri inviati da script.

Poiché l'invocazione dinamica è un argomento complesso e di uso non comune, verrà proposto un semplice esempio che non utilizzi l'IR, ma sia pilotato da linea di comando.

Si definisca una semplice interfaccia come segue

```
// IDL
module dii {
   interface DynObject {
      string m0();
      string m1(in string p1);
      string m2(in string p1, in string p2);
      string m3(in string p1, in string p2, in string p3);
   };
};
```

Come si è detto in precedenza, l'uso di DII non ha alcuna influenza sul lato server. Il server si limiterà a istanziare l'oggetto e a registrarlo col nome di `DynObject` sul Naming Service.

L'implementazione dell'oggetto è molto semplice e ha come unico scopo quello di consentire il controllo dei parametri e del metodo invocato

```
package server;

import dii.*;

public class DynObjectImpl extends _DynObjectImplBase {

   public DynObjectImpl() {
      super();
   }

   public String m0() {
      return "Metodo 0 # nessun parametro";
   }

   public String m1(String p1) {
      return "Metodo 1 # " + p1;
   }

   public String m2(String p1, String p2) {
      return "Metodo 2 # " + p1 + " - " + p2;
   }

   public String m3(String p1, String p2, String p3) {
```

```
        return "Metodo 3 # " + p1 + " - " + p2 + " - " + p3;
    }
}
```

Il client deve identificare dall'input da linea di comando il metodo da invocare e i suoi eventuali parametri; il primo passo significativo da compiere è l'accesso al Naming Service.

```
//...
org.omg.CORBA.Object obj = ncRef.resolve(path);
```

Poiché stub, skeleton, Helper e Holder saranno distribuiti solo sul server, non è possibile effettuare un narrow.

A questo punto è possibile iniziare a costruire la Request con il metodo più semplice

```
org.omg.CORBA.Request request = obj._request(args[0]);
```

Si noti che il parametro passato `args[0]` è il nome del metodo che si intende utilizzare, letto da linea di comando.

Ora è possibile costruire la lista di argomenti; nel caso in esame la lista è costruita dinamicamente effettuando un parsing dell'array di input.

```
org.omg.CORBA.NVList arguments = request.arguments();

for (int i = 1; i < args.length; i++) {
    org.omg.CORBA.Any par = orb.create_any();
    par.insert_string(args[i]);
    arguments.add_value("p" + 1, par, org.omg.CORBA.ARG_IN.value);
}
```

Ogni valore della `NVList` è rappresentato da un oggetto di tipo `Any`; questo è uno speciale tipo CORBA che può incapsulare qualunque altro tipo e ha un'API che fornisce specifiche operazioni di inserimento ed estrazione di valori (nell'esempio si usano `insert_string` ed `extract_string`).

Per invocare il metodo è ancora necessario impostare il tipo di ritorno. Per fare questo si utilizza l'interfaccia `TypeCode` che è in grado di rappresentare qualunque tipo IDL. Le costanti che identificano i `TypeCode` dei vari tipi IDL sono fornite da `TCKind`.

```
request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string));

request.invoke();
```

L'invocazione utilizza la normale chiamata sincrona di metodo data da `invoke`. Terminata l'invocazione è possibile ottenere e visualizzare la stringa di ritorno.

```
org.omg.CORBA.Any method_result = request.return_value();
System.out.println(method_result.extract_string());
```

L'esecuzione dell'esempio è possibile con VisiBroker utilizzando i consueti passi, in particolare si faccia riferimento all'esempio visto in precedenza sul Naming Service.

DII fornisce un meccanismo estremamente elastico e flessibile che prospetta un sistema assolutamente libero in cui tutti gli oggetti interrogano un Trader Service, individuano e utilizzano dinamicamente i servizi di cui necessitano.

Malauguratamente l'invocazione dinamica presenta alcuni limiti che ne condizionano l'utilizzo. In primo luogo l'invocazione dinamica è decisamente più lenta dell'invocazione statica poiché ogni chiamata a metodo implica un gran numero di operazioni remote.

In secondo luogo la costruzione di Request è un compito complesso e richiede uno sforzo supplementare in fase di sviluppo (lo sviluppatore deve implementare i compiti normalmente svolti dallo stub). L'invocazione dinamica è inoltre meno robusta in quanto l'ORB non può effettuare il typechecking prima dell'invocazione, questo può causare anche un crash durante l'unmarshalling.

Callback

Esistono situazioni in cui un client è interessato al verificarsi di un particolare evento sul server (cambiamento di uno stato, occorrenza di un dato errore, …). In un ambiente distribuito, in cui tipicamente non è possibile conoscere a priori il numero di client, soddisfare questa necessità è problematico.

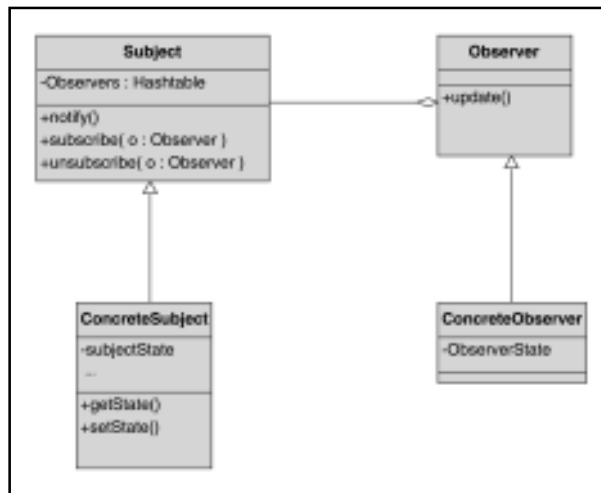
Il meccanismo più semplice implementabile è il polling: il client periodicamente controlla il valore cui è interessato. Questo approccio è poco robusto e implica un notevole spreco di risorse. Approcci migliori sono forniti da meccanismi di notifica asincrona quali CORBA Event Service o sistemi di messaggistica.

Un'ultima possibilità è quella di utilizzare una callback: sarà il server a invocare direttamente un metodo di notifica sul client o su un oggetto ad esso associato. Questo approccio ha notevoli vantaggi computazionali, ma può presentare difficoltà nel caso in cui la topologia di rete veda il client e il server separati da un firewall.

Tralasciando momentaneamente i problemi di rete, l'utilizzo di callback si presta bene a risolvere alcuni classici problemi di programmazione distribuita. A titolo d'esempio s'immagini la costruzione di una semplice chat.

I client devono ricevere una notifica nel caso in cui si connetta un nuovo utente, venga inviato un messaggio o un utente lasci la chat. Si può organizzare l'applicazione adottando un meccanismo simile a quello della gestione degli eventi Java: i client sottoscrivono presso il server chat un servizio di notifica.

La soluzione classica per problemi di questo tipo è data dal pattern Observer. Questo pattern è pensato proprio per quelle situazioni in cui un cambiamento di stato su un

Figura 16.9 – *Il pattern Observer*

oggetto (Subject) ha potenziali impatti su un numero imprecisato di altri oggetti (Observers). In queste situazioni solitamente è necessario inviare agli Observers una generica notifica.

Il numero di Observers deve poter variare a runtime e il Subject non deve conoscere quali informazioni sono necessarie al generico Observer. Per questa ragione la formulazione classica del pattern prevede che siano definiti:

- una classe astratta `Subject` che fornisca i metodi `subscribe`, `unsubscribe` e `notify`;
- una classe `ConcreteSubject` che definisca i metodi di accesso alle proprietà interessate;
- una classe `Observer` che fornisca il metodo di ricezione notifica;
- una classe `ConcreteObserver` che mantenga un riferimento al `ConcreteSubject` e fornisca il metodo `update` utilizzato per riallineare il valore delle proprietà.

Quando un Subject cambia stato invia una notifica a tutti i suoi Observer (quelli che hanno invocato su di lui il metodo `subscribe`) in modo tale che questi possano interrogare il Subject per ottenere le opportune informazioni e riallinearsi al Subject.

Il pattern Observer, anche conosciuto come Publish/Subscribe, è molto utilizzato nei casi in cui è necessario implementare meccanismi "1 a *n*" di notifica asincrona. Anche il CORBA Event Service adotta il pattern Observer.

Tornando alla chat, l'implementazione del pattern può essere leggermente semplificata facendo transitare direttamente con la notifica il messaggio cui i client sono interessati.

Nell'esempio in esame ogni client si registrerà come Observer presso il server chat (in uno scenario reale probabilmente si definirebbe un oggetto `Observer` separato). Il server invocherà il metodo di notifica su tutti gli Observer registrati inviando loro il messaggio opportuno. Nello scenario in esame anche l'oggetto `Observer`, essendo remoto, dovrà essere attivato sull'ORB e definito via IDL.

```
// IDL
module chat {

   // Forward declaration
   interface SimpleChatObserver;

   struct User {
      string userId;
      SimpleChatObserver callObj;
   };

   struct Message {
      User usr;
      string msg;
   };

   // OBSERVER
   interface SimpleChatObserver {
      // N.B. Il server non aspetta alcuna
      // risposta dai vari client
      oneway void callback(in Message msg);
   };

   // SUBJECT
   interface SimpleChat {
      void subscribe(in User usr);
      void unsubscribe(in User usr);
      void sendMessage(in Message msg);
   };

};
```

Nell'esempio non vengono utilizzati gli adapter e l'attivazione viene effettuata mediante `connect` (quindi è utilizzabile anche con Java IDL). Sarà il client stesso a registrarsi presso l'ORB.

```
package client;

import chat.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient extends _SimpleChatObserverImplBase {

   private SimpleChat chat = null;
   private User user = null;
   private JTextField tMsg = new JTextField();
   private JButton bSend = new JButton("Send");
   private JTextArea taChat = new JTextArea();

   public SimpleChatClient() {
      super();

      // qui il codice di inizializzazione UI
   }

   public void init(String userId) throws Exception {

      // Crea e inizializza l'ORB
      ORB orb = ORB.init((String[])null, null);

      // Root naming context
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Utilizzo il Naming per ottenere il riferimento
      NameComponent nc = new NameComponent("SimpleChat", " ");
      NameComponent path[] = {nc};
      chat = SimpleChatHelper.narrow(ncRef.resolve(path));

      // Si registra presso l'ORB
      // N.B. Il server deve essere in grado di effettuare
      // un'invocazione remota del metodo callback
      orb.connect(this);

      // Crea e registra user
      user = new User(userId, this);
      chat.subscribe(user);
   }

   // Metodo remoto di notifica invocato dal server
```

```
   public void callback(Message msg) {
      taChat.append("#" + msg.usr.userId + " - " + msg.msg + "\n");
      tMsg.setText(" ");
   }

   // Lo userId del client è passato da linea di comando
   public static void main(String args[]) throws Exception {
      SimpleChatClient sc = new SimpleChatClient();
      sc.init(args[0]);
   }
}
```

Il riferimento al client viene fatto transitare nell'oggetto `User` e registrato presso la chat dal metodo `subscribe`. Per semplicità la deregistrazione del client è associata all'evento di chiusura della finestra (in un contesto reale sarebbe necessario un approccio più robusto).

```
JFrame f = new JFrame("SIMPLE CHAT");
f.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      try {
         chat.unsubscribe(user);
         System.exit(0);
      } catch (Exception ex) {}
   }
});
```

L'oggetto `SimpleChatImpl` è invece avviato e registrato presso l'ORB da un oggetto server con le modalità consuete. `SimpleChatImpl` definisce il funzionamento della chat vera e propria.

```
package server;

import chat.*;
import java.util.Hashtable;
import java.util.Enumeration;

public class SimpleChatImpl extends _SimpleChatImplBase {

   Hashtable h = new Hashtable();

   public SimpleChatImpl() {
      super();
   }

   // Aggiunge un utente alla chat
```

```
   public synchronized void subscribe(User user) {
      h.put(user.userId, user);

      Message msg = new Message(user, " has joined the chat");
      this.sendMessage(msg);

      System.out.println("Added: " + user.userId);
   }

   // Rimuove un utente dalla chat
   public synchronized void unsubscribe(User user) {
      h.remove(user.userId);

      Message msg = new Message(user, " left the chat");
      this.sendMessage(msg);

      System.out.println("Removed: " + user.userId);
   }

   // Invia il messaggio a tutti gli utenti
   public void sendMessage(Message msg) {
      User user = null;

      for (Enumeration e = h.elements(); e.hasMoreElements();) {
         user = (User) e.nextElement();

         // Invoca la callback
         try {
            user.callObj.callback(msg);
         } catch (Exception ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

Figura 16.10 – *Il client chat in esecuzione*

L'esecuzione dell'esempio segue i consueti passi. Come detto in precedenza, un possibile problema è legato all'eventuale presenza di firewall che inibiscano l'invocazione da server a client.

CORBA e i firewall

Per ragioni di sicurezza tutte le Intranet aziendali utilizzano firewall per limitare il traffico di rete da e verso un certo host. Il firewall agisce come una barriera bidirezionale limitando il traffico sulla base di valutazioni sull'origine/destinatario della richiesta e sul tipo di protocollo utilizzato.

Ogni tecnologia di comunicazione distribuita deve quindi fare i conti con l'eventuale presenza di firewall. Il problema è complesso sia perché non esiste uno standard realmente accettato, sia perché la presenza di un firewall può riguardare tanto il lato server quanto il lato client.

Una tipica configurazione aziendale limita il traffico alla porta 80 riservata al traffico HTTP. Per questa ragione una soluzione comune, adottata anche da RMI, è quella di incapsulare il traffico client/server, JRMP o IIOP, in pacchetti HTTP request/response (HTTP tunneling). In questo modo le richieste/risposte possono attraversare il firewall come le normali attività Web.

Nel caso si utilizzi il tunneling, il Web Server deve essere in grado di riconoscere il particolare tipo di richiesta, estrarre i dati dal pacchetto HTTP e ridirigere la richiesta verso la risorsa opportuna. In questo modo il Web Server agisce sostanzialmente da router.

Per effettuare il routing, il Web Server utilizza CGI o Servlet con ovvi impatti negativi sulle performance. Inoltre, data la natura unidirezionale di HTTP, adottando il tunneling non è possibile utilizzare invocazioni da server a client (notifiche di eventi e callback in genere).

Per queste ragioni il tunneling non si è mai affermato nel mondo CORBA e storicamente ad esso sono sempre state preferite soluzioni alternative proprietarie. Le più celebri soluzioni software fornite dal mercato sono Inprise GateKeeper e IONA WonderWall legate agli ORB VisiBroker e OrbixWeb.

Il funzionamento dei due prodotti è similare: ambedue lavorano come Proxy IIOP tra client e server che abbiano restrizioni di security tali da impedire loro di comunicare direttamente. In modo trasparente al client si occupano di ricevere, controllare e "forwardare" il traffico IIOP da e verso l'oggetto remoto opportuno. Tutte le informazioni necessarie al forward transitano con i reference CORBA. Poiché lavorano in ambedue le direzioni, non presentano i limiti della soluzione tunneling sull'utilizzo di callback.

Opportunamente utilizzati questi prodotti consentono anche di utilizzare CORBA con Applet non certificate. Le restrizioni date dalla sandbox non consentono infatti comunicazioni con host differenti da quello da cui l'Applet è stata scaricata e questo è in evidente contrasto con il principio della location transparency CORBA.

Ponendo Gatekeeper o WonderWall sullo stesso host del Web Server, la comunicazione con l'Applet non violerà i limiti della sandbox. A livello TCP l'applet comunicherà direttamente con un oggetto proxy contenuto nei due prodotti. Sarà questo oggetto a effettuare l'invocazione reale al servant CORBA e a gestire in maniera similare la risposta.

Prodotti come quelli citati forniscono soluzioni decisamente superiori al tunneling, ma non appartengono allo standard. CORBA 3.0 ha finalmente fornito specifiche esaustive per i GIOP Proxy Firewall.

Nel caso sia consentito è comunque possibile aprire una porta sul firewall e vincolare le applicazioni CORBA a comunicare soltanto su di essa (le modalità variano da ORB a ORB).

CORBA e J2EE

Tipicamente CORBA viene presentato come middleware alternativo alle altre tecnologie per lo sviluppo di applicazioni distribuite: RMI, EJB, DCOM, ecc.

Poiché però le specifiche danno grande risalto alle necessità di integrazione poste dalle applicazioni Enterprise, CORBA consente in varia misura di interagire con le tecnologie di cui sopra.

Nel caso di DCOM le possibilità di integrazione pongono alcuni problemi e tipicamente si limitano all'utilizzo di bridge che convertono le invocazioni CORBA in opportune invocazioni DCOM.

Completamente differente è invece il ruolo che CORBA si trova a giocare nella piattaforma J2EE. Grazie agli sforzi congiunti di OMG e Sun sono state definite alcune specifiche (object-by value, RMI/IDL, EJB 1.1, CORBA/EJB interop) che non si limitano a portare CORBA a buoni livelli di interoperabilità con la piattaforma Java Enterprise, ma ne fanno un fondamentale elemento infrastrutturale.

La piattaforma J2EE fornisce inoltre funzionalità di chiara ispirazione CORBA. Java Naming e Directory Interface sono correlate al CORBA Naming Service. Java Transaction API e Java Transaction Service sono correlate al CORBA Object Transaction Service. Molte similitudini possono essere inoltre individuate in altri ambiti J2EE: gestione della Security, gestione della persistenza e utilizzo di *Messaging Oriented Middleware* (MOM).

CORBA vs RMI

Come si è avuto modo di osservare esistono notevoli somiglianze tra l'utilizzo di CORBA e quello di RMI. In realtà è evidente che molte delle conclusioni e dei dettami OMG sono confluiti nella definizione dell'architettura RMI.

Ad alto livello l'utilizzo di RMI è più semplice di quello CORBA. La ragione di questo va ricercata nell'implicita semplificazione che si ha dovendo fornire specifiche monolinguaggio. Inoltre RMI può contare su un notevole insieme di funzionalità base

offerte dal linguaggio Java e non supportate dagli altri linguaggi utilizzabili con CORBA (si pensi alla serializzazione e all'introspezione).

In generale CORBA può offrire performance superiori a RMI e migliori soluzioni di clustering e load balancing. Le differenze possono essere più o meno significative a seconda dell'ORB utilizzato.

La scelta tra l'utilizzo dell'una o dell'altra tecnologia è quindi legata alle esigenze d'utilizzo. In uno scenario semplicemente Java senza alcuna necessità di integrazione con altri linguaggi la scelta RMI potrà essere accettabile. In uno scenario Enterprise eterogeneo o con significative esigenze di tuning e performance sarà da preferire l'adozione di CORBA.

Sebbene CORBA e RMI vengano spesso presentate come tecnologie alternative, esiste la concreta possibilità di farli cooperare utilizzando RMI con il protocollo CORBA IIOP.

RMI-IIOP

L'utilizzo tipico di RMI prevede l'adozione del protocollo di trasporto proprietario *Java Remote Method Protocol* (JRMP). Nel 1998 specifiche prodotte da Sun e IBM hanno introdotto la possibilità di utilizzare RMI sul protocollo IIOP. L'elemento fondamentale di RMI-IOOP è costituito dalle specifiche Object-by-value che consentono di adottare con CORBA il passaggio di parametri per valore tipico di RMI.

La definizione di RMI-IIOP fornisce allo sviluppatore i vantaggi di semplicità RMI uniti alle caratteristiche di portabilità/interoperabilità del modello CORBA. La soluzione è adottata anche dall'infrastruttura EJB.

Per lo sviluppatore Java si ha inoltre il vantaggio di non dover imparare il linguaggio IDL. La definizione delle interfacce viene operata direttamente in Java con modalità RMI. A partire dalle interfacce Java un compilatore apposito genererà tutte le classi di infrastruttura CORBA.

Con VisiBroker vengono forniti due compilatori: java2iiop si occupa di generare stub, skeleton, Helper e Holder, java2idl consente di ottenere la rappresentazione IDL.

Anche il JDK fornisce un completo supporto per RMI/IIOP. Utilizzando il JDK 1.3 questo è già fornito con l'SDK, mentre nel caso si utilizzi un JDK precedente sarà necessario scaricarlo a parte insieme ai tool RMI/IIOP (si veda [RMI over IIOP] in bibliografia). In ogni caso andrà utilizzata la nuova versione del compilatore rmic con i flag `-iiop` o `-idl`.

Le specifiche EJB 1.1 indicano RMI-IIOP come API standard di comunicazione. L'uso di RMI-IIOP è la chiave della compatibilità tra CORBA ed EJB. Già prima di queste specifiche alcuni Application Server fornivano il layer EJB sopra una infrastruttura CORBA (Inprise Application Server, ecc.). Altri produttori utilizzavano invece implementazioni proprietarie dell'API RMI (Bea WebLogic, ecc.).

Bibliografia

La homepage di OMG: `http://www.omg.org`

La specifica: `ftp://ftp.omg.org/pub/docs/ptc/96-03-04.pdf`

Java IDL: `http://java.sun.com/products/jdk/idl`

[VisiBroker]: `http://www.inprise.com/visibroker/`

[prodotti CORBA]:
`http://www.infosys.tuwien.ac.at/Research/Corba/software.html`

[RMI over IIOP]: `http://java.sun.com/products/rmi-iiop/index.html`

Enterprise Java Beans

DI GIOVANNI PULITI

La specifica di Enterprise Java Beans definisce un modello di tipo CTM (Component Transaction Monitors), uno dei modelli più avanzati nel mondo delle business application.

Da un punto di vista architetturale EJB è semplicemente un modo diverso di vedere e di implementare il modello della computazione distribuita, della quale nel corso degli anni si sono viste numerose variazioni sul tema.

Fra le molte soluzioni proposte negli ultimi tempi, una delle più famose è quella basata sugli oggetti distribuiti, i quali possono essere eseguiti in remoto e su più macchine indipendenti fra loro.

Già prima di EJB erano disponibili alcune interessanti ed importanti alternative: da un lato Remote Method Invocation (RMI), permette in maniera piuttosto semplice di realizzare applicazioni distribuite e rappresenta la soluzione full-Java più semplice anche se probabilmente meno potente e flessibile.

In alternativa, CORBA è spesso considerato il più versatile, eterogeneo, potente — ma anche più complesso — sistema di gestione degli oggetti remoti e distribuiti.

Non si prenderanno in considerazione tecnologie come MTS o DCOM di Microsoft dato che, sulla base della filosofia adottata in questo testo, non possono essere ritenute esempi di tecnologia distribuita, limitando il loro utilizzo alla sola piattaforma Windows.

La differenza fondamentale di EJB rispetto agli altri modelli è rappresentata dal supporto offerto per la realizzazione di applicazioni distribuite: RMI e CORBA infatti sono esclusivamente motori di oggetti distribuiti (solo Java nel primo caso, *language neutral* nel secondo), dove i cosiddetti distributed object services devono essere implementati a mano o forniti per mezzo di software terze parti.

In EJB invece tutto quello che riguarda la gestione dei servizi è disponibile ed utilizzabile in maniera automatica e trasparente agli occhi del client.

Per servizi si intendono quelle funzionalità atte alla implementazione dei tre aspetti qui di seguito descritti.

Transazioni

Sinteticamente una transazione è definibile come un insieme di operazioni che devono essere svolte in maniera atomica, oppure fallire totalmente. La sola atomicità non però è un requisito sufficiente a garantire la correttezza delle operazioni svolte e la coerenza dei dati memorizzati: come si avrà modo di vedere in seguito, per definire il livello di bontà di un sistema transazionale spesso ci si riferisce all'acronimo ACID.

La transazionalità è molto importante nel mondo del commercio elettronico e nella maggior parte delle casistiche reali. I container EJB offrono un sistema automatico per la gestione delle transazioni.

Security

Java è nato come linguaggio estremamente attento al concetto di sicurezza, aspetto ancora più importante quando si ha a che fare con sistemi distribuiti. Nella specifica EJB è stato inserito un supporto integrato alla security basandosi su quello offerto dalla piattaforma Java 2.

Scalabilità

La filosofia con cui EJB è stato progettato, e soprattutto la modalità con cui opera, consentono un elevato livello di flessibilità e scalabilità in funzione del traffico dati e del numero di clienti. Ad esempio la politica per la decisione di quali, quanti e come i vari bean vengano forniti ai vari client è totalmente gestita dal server.

Da notare che EJB è costruito sopra il modello di RMI del quale utilizza le funzionalità e la filosofia di base per realizzare strutture distribuite, estendendone come visto i servizi in maniera trasparente.

Architettura

La specifica di EJB definisce una architettura standard per l'implementazione della business logic in applicazioni multiTier basate su componenti riusabili. Tale architettura è fondata essenzialmente su tre componenti: il server, i container e i vari client.

Server EJB

Il server EJB ha lo scopo di incapsulare tutto quello che sta sotto lo strato EJB (applica-

zioni legacy, servizi distribuiti) e di fornire ai contenitori gli importanti servizi di base per i vari componenti installati.

Contenitore

Intercettando le invocazioni dei client sui metodi dei bean, il contenitore è in grado di effettuare diversi controlli molto importanti, in modo da offrire alcune funzionalità legate al life cycle dei vari componenti, alla gestione delle transazioni ed alla security management.

Dal punto di vista implementativo, al fine di permettere la corretta comunicazione fra componente e contenitore, chi sviluppa un componente deve implementare alcune interfacce standard.

Client

Il client infine rappresenta l'utilizzatore finale del componente. La visione che ha del componente è resa possibile grazie a due interfacce (`HomeInterface` e `RemoteInterface`), la cui implementazione è effettuata dal container al momento del deploy del bean.

La `Home Interface` fornisce i metodi relativi alla creazione del bean, mentre l'altra offre al client la possibilità di invocare da remoto i vari metodi di business logic. Implementando queste due interfacce, il container è in grado di intercettare le chiamate provenienti dal client e al contempo di fornire ad esso una visione semplificata del componente. Il client non ha la percezione di questa interazione da parte del container: il vantaggio è quindi quello di offrire in modo indolore la massima flessibilità, scalabilità oltre a una non indifferente semplificazione del lavoro da svolgere.

Tipi di Enterprise Java Beans

Nella versione 1.1 della specifica di EJB sono stati definiti due tipi di componenti, gli entity beans e i session beans. A partire dalla 2.0 sarà possibile disporre di un terzo tipo di componente, orientati alla gestione di sistemi di messaggistica.

Un entity bean rappresenta un oggetto fisico, un concetto rappresentabile per mezzo di una parola: per questo motivo spesso rappresentano i soggetti in gioco nella applicazione, anche se generalmente svolgono un ruolo passivo, essendo utilizzati prevalentemente dal client.

Gli entity sono in genere mantenuti persistenti per mezzo del supporto di un database di qualche tipo.

I session bean invece possono essere visti come un'estensione della applicazione client: sono responsabili della gestione della applicazione o dei vari task, ma non dicono niente del contesto all'interno del quale tali operazioni sono prese.

Ad esempio prendendo in considerazione il caso dell'organizzazione di un corso su Java, si può immaginare che l'aula stessa sia rappresentabile con un entity bean, così come i vari PC, gli oggetti che compongono l'aula, il docente e i partecipanti stessi. Tutto quello che invece riguarda la coordinazione delle varie risorse, la prenotazione delle aule e la gestione degli alunni, così come lo scambio di informazioni e messaggi è un tipo di lavoro svolto da un session bean.

Strutturazione dei vari componenti

Prima di procedere oltre nella analisi della architettura di un bean, può risultare utile vedere cosa sia necessario fare per crearne uno partendo dalla definizione delle sue componenti.

Per prima cosa, è necessario implementare due classi ed estendere due interfacce. Per comodità spesso ci si riferisce al concetto di bean come tutto l'insieme degli oggetti e delle interfacce che compongono il bean: il nome utilizzato è quello che fa riferimento alla bean class. Di seguito se ne riportano le caratteristiche.

Remote Interface

Per prima cosa un bean deve definire una interfaccia remota tramite l'estensione della interfaccia `javax.ejb.EJBObject`, la quale deriva a sua volta dalla ben nota `java.rmi.Remote`. I metodi esposti della interfaccia remota costituiscono i cosiddetti business methods del bean, ovvero le funzionalità che il client sarà in grado di utilizzare da remoto. L'oggetto che implementerà tale interfaccia e che viene creato dal server al momento del deploy, viene detto in genere `EJBObject`.

Home Interface

Questa interfaccia invece definisce il set di metodi necessari per gestire il ciclo di vita del componente: tali metodi sono invocati dal container durante le fasi di creazione ed installazione nel container, durante la rimozione, o in corrispondenza di operazioni di ricerca per mezzo di chiavi univoche.

L'interfaccia da estendere in questo caso è la `javax.ejb.EJBHome`, derivante anch'essa da `java.rmi.Remote`, ed è per questo motivo che caso l'interfaccia creata viene detta EJBHome o più semplicemente home interface.

Bean Class

Per costruire un entity bean va definita una classe derivante da `javax.ejb.EntityBean`, mentre per i session bean l'interfaccia da estendere è la `javax.ejb.SessionBean`.

Questa classe contiene la business logic del bean: essa non deve implementare le due interfacce remote appena viste, anche se suoi metodi devono corrispondere a quelli definiti nella remote interface ed in qualche modo sono associabili a quelli specificati nella home interface.

Il fatto che per creare un bean si debbano estendere delle interfacce che poi sono implementate dal container al momento del deploy, è la base della filosofia di EJB: infatti oltre alla notevole semplificazione che questo comporta, tale sistema consente l'interazione fra il client e l'oggetto remoto in modo indiretto, tramite l'intermediazione del container, il quale si preoccupa di creare nuove istanze di oggetti remoti e di verificarne la corretta installazione e il giusto funzionamento, e così via.

Primary Key

Normalmente ogni bean, specie nel caso degli entity, è rappresentabile per mezzo dei valori delle variabili in esso contenuto. Tali proprietà finiscono spesso per diventare la chiave (semplice o composta da più campi) tramite la quale identificare univocamente il bean.

Nel caso in cui il bean sia particolarmente complesso, o i dati non utilizzabili direttamente, si può procedere a definire una classe apposita che svolgerà il ruolo di chiave per l'oggetto. In genere questa classe verrà utilizzata dai vari metodi di ricerca, messi a disposizione dalla interfaccia home, come si avrà modo di vedere in seguito.

Implementazione di un entity bean

Al fine di comprendere meglio come strutturare le varie parti di un componente, si riconsideri il caso relativo alla gestione di corsi Java, analizzando cosa sia necessario fare per implementare un entity bean; il caso dei session bean non è molto differente.

La struttura base delle classi e delle interfacce necessarie per mettere in piedi lo scheletro di una applicazione di questo tipo è riportata nelle figg. 17.1 e 17.2.

Il codice per la remote interface è il seguente

```
public interface Course extends EJBObject {
   public String getCourseName();
   public int getType();
   public void setCourseName(String name);
   public void setType(int type);
}
```

mentre per la home interface si ha

```
public interface CourseHome extends EJBHome {
   public CourseBean create() throws CreateException, RemoteException ;
```

Figura 17.1 – *Organizzazione gerarchica di remote e home interface*

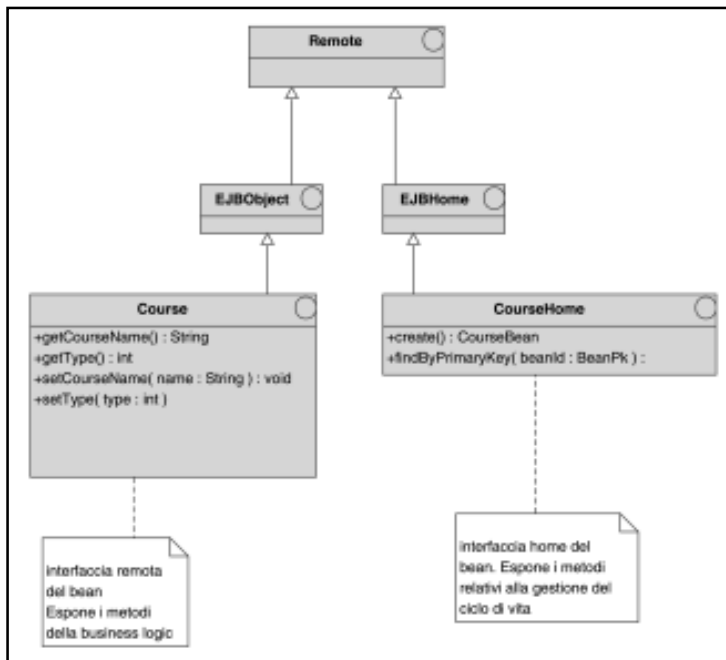
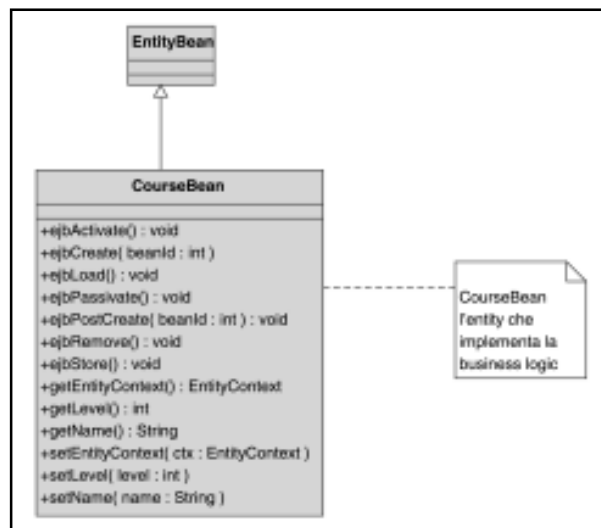



Figura 17.2 – *La classe* `CourseBean`*, che implementa la business logic*

```
   public Course findByPrimaryKey(BeanPK id)  throws  FinderException,
                                                      RemoteException;
}
```

Il codice corrispondente alla classe `CourseBean`

```
public class CourseBean extends EntityBean{
   // metodi creazionali
   public void ejbActivate() { }
   public void ejbCreate(int id) { }
   public void ejbLoad() { }
   public void ejbPassivate() { }
   public void ejbPostCreate(int id) { }
   public void ejbRemove() { }
   public void ejbStore() { }

   // metodi per la gestione del context
   public EntityContext getEntityContext() {
      return null;
   }
   public void setEntityContext(EntityContext ctx) { }

   // business methods
   public int getLevel() throws RemoteException {
      return 0;
   }
   public String getName() throws RemoteException {
      return null;
   }
   public void setLevel() throws RemoteExcpetion { }
   public void setName(String name) throws RemoteException { }
}
```

Per prima cosa si possono individuare i vari metodi relativi al ciclo di vita del componente, invocati in modalità di callback dal container: essi sono necessari per il modello EJB, ma non fanno parte dell'interfaccia pubblica del componente; inoltre essendo parte integrante della interfaccia `EntityBean`, e quindi delle specifiche del modello EJB, ne è obbligatoria la definizione ma non l'implementazione.

Al loro interno si potranno inserire tutte le operazioni che si desidera siano effettuate durante una determinata fase del ciclo di vita del bean, senza che ci si debba preoccupare di quando e come essi verranno invocati, ovvero tralasciando i problemi legati a come e quando verranno effettuati i passaggi di stato sul componente.

Possono però essere pensati come sistema di notifica sul bean di un imminente passaggio di stato nell'ambito del suo ciclo di vita.

L'adozione di questa architettura permette di operare una netta separazione fra il lavoro di chi sviluppa il bean (che non ha né la necessità né la volontà di sapere come sia

implementato il server, potendosi concentrare esclusivamente sui vari business methods del componente) e chi sviluppa il server (che non deve conoscere, né oltretutto potrebbe, i dettagli implementativi delle varie applicazioni distribuite basate sul modello EJB).

I metodi `ejbCreate()` ed `ejbPostCreate()` sono invocati dal container sul componente rispettivamente subito prima e subito dopo la creazione del componente, mentre il metodo `ejbRemove()` comunica al componente che sta per essere rimosso dal server, e che i dati relativi, memorizzati nel db, stanno per essere cancellati.

Invece `ejbLoad()` ed `ejbStore()` notificano che lo stato del componente sta per essere sincronizzato con il database. In particolare il metodo `ejbStore()` viene chiamato dal container subito prima che il componente venga memorizzato nel database, e tra le altre cose permette al programmatore di effettuare tutte le operazioni di sincronizzazione e messa a punto prima che il componente stesso sia reso persistente.

Analogamente l'invocazione di `ejbLoad()` avviene immediatamente dopo la fase di caricamento del componente dal db.

I metodi `ejbActivate()` e `ejbPassivate()` notificano al componente che sta per essere attivato o disattivato.

Gli altri metodi presenti nel bean sono i cosiddetti accessori e mutatori (`setXXX` e `getXXX`) e permettono di gestire le proprietà dell'oggetto remoto: insieme ai business methods sono visibili dal client, e rappresentano perciò l'interfaccia pubblica vera e propria del componente.

Per la ricerca di una particolare istanza di bean da parte del client, si deve provvedere alla definizione di almeno un metodo di ricerca per chiave univoca: il `findByPrimaryKey()` è il metodo che necessariamente deve essere presente nella home interface, ma si possono aggiungere altri metodi che consentano di reperire i componenti tramite altri criteri particolari. I seguito tali concetti saranno ripresi ed approfonditi.

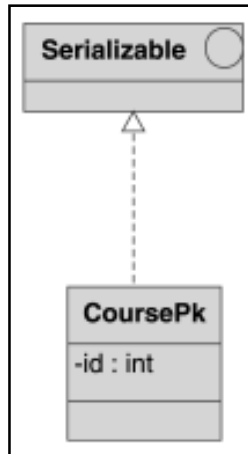
La chiave primaria deve essere definita dallo sviluppatore del bean, e deve essere di un tipo serializzabile. Ad esempio si può pensare di definire una chiave nel seguente modo

```
public class CoursePK implements Serializable {
   private int id;
}
```

il cui grafico UML è riportato in fig. 17.3.

Implementazione di un Session Bean

Per quanto riguarda la creazione di un session bean, il procedimento, con le dovute modifiche, è sostanzialmente analogo al caso degli entity. Innanzitutto deve essere implementata l'interfaccia `javax.ejb.SessionBean` invece della `EntityBean`. Questo comporta la presenza del metodo `ejbCreate()` ma non di `ejbPostCreate()`.

Figura 17.3 – *La classe* `CoursePK`*, che implementa l'interfaccia* `Serializable`*, rappresenta la chiave univoca del bean identificato in questo caso da un valore intero*

Inoltre, dato che un bean di questo tipo non è persistente, non sono presenti i metodi `ejbLoad()` ed `ejbStore()`. Similmente è presente il metodo `setSessionContext()` ma non `unsetSessionContext()`.

Infine un session bean dovrebbe fornire un metodo `ejbRemove()` da utilizzare per avvertire il bean quando il client non ne ha più bisogno: i questo caso però, differentemente da ciò che accade con un entity, non vi è nessuna rimozione dei dati dal database, dato che i session bean non prevedono nessun meccanismo di memorizzazione persistente.

Sempre per lo stesso motivo i session bean non sono associati ad una chiave primaria e quindi non è presente nessun metodo di ricerca.

EJB dietro le quinte

Non è stato detto niente fino ad ora relativamente al processo si implementazione delle due interfacce remote ed home. Per non lasciare troppo in sospeso gli aspetti legati alla gestione degli oggetti remoti, è bene fare una analisi leggermente più approfondita sia su `EJBObject` che su `EJBHome`. Tali interfacce sono implementate dal container al momento del deploy, tramite la creazione di un oggetto remoto che funziona come wrapper per il bean stesso (ovvero per la classe che implementa ad esempio l'interfaccia `EJBEntity`).

Il modo con cui tale oggetto viene creato dipende molto dall'implementazione del server, ma si basa sulle informazioni contenute nel bean e nei vari descriptor files: esso lavora in stretto contatto con il container e provvede a fornire tutte le funzionalità di gestione delle transazioni, di security ed altre funzionalità a livello di sistema.

La configurazione più utilizzata dal server è quella in cui l'oggetto remoto implementa l'interfaccia remota e contiene un riferimento alla classe che implementa l'entity interface.
Questa soluzione è rappresentata schematicamente in fig. 17.4. Il secondo schema invece prevede una soluzione mista, in cui l'oggetto remoto estende da una parte il bean, mentre dall'altra implementa l'interfaccia remota (fig. 17.5). Infine il terzo schema (fig. 17.6), si basa su una configurazione particolare, che di fatto non prevede la partecipazione del bean creato dallo sviluppatore. In questo caso è l'oggetto remoto che ha una implementazione proprietaria che rispecchia da un lato il server remoto, dall'altro i metodi del bean.

Anche la EJB Home viene generata automaticamente dal server, al momento dell'installazione del bean, quando vengono implementati tutti i metodi definiti nella home interface.

Ad esempio quando viene invocato il metodo `create()` sulla home interface, l'EJBHome crea una istanza dell' EJBObject, fornendogli una istanza di bean del tipo opportuno. Nel caso di entity bean, viene anche aggiunto un nuovo record nel database.

Quando il metodo `ejbCreate()` ha terminato, l'EJBObject restituisce uno stub dell'oggetto remoto al client.

Figura 17.4 – *Architettura di organizzazione delle classi remote e del bean secondo la configurazione tipica a wrapper*

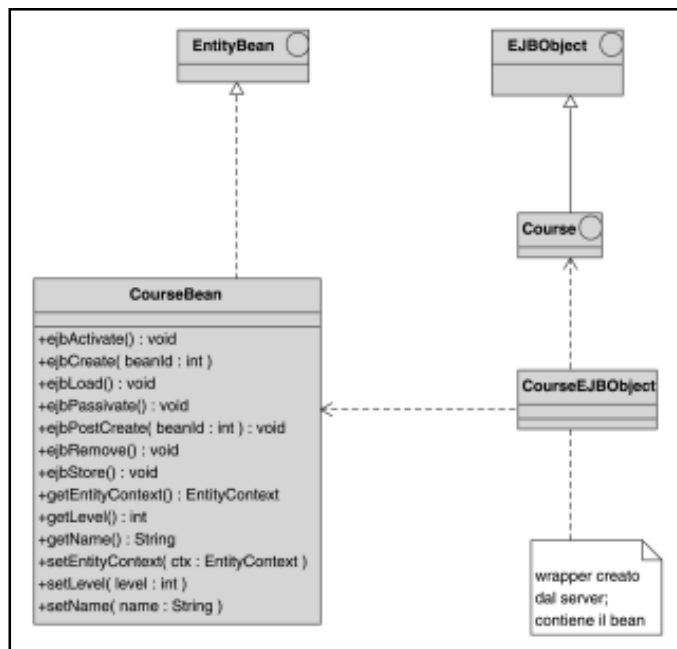



Figura 17.5 – *Architettura mista in cui il* `CourseEJBObject` *implementa il* `CourseBean` *e l'interfaccia* `Course`*. Questo caso può essere considerato come una variante del classico wrapper*

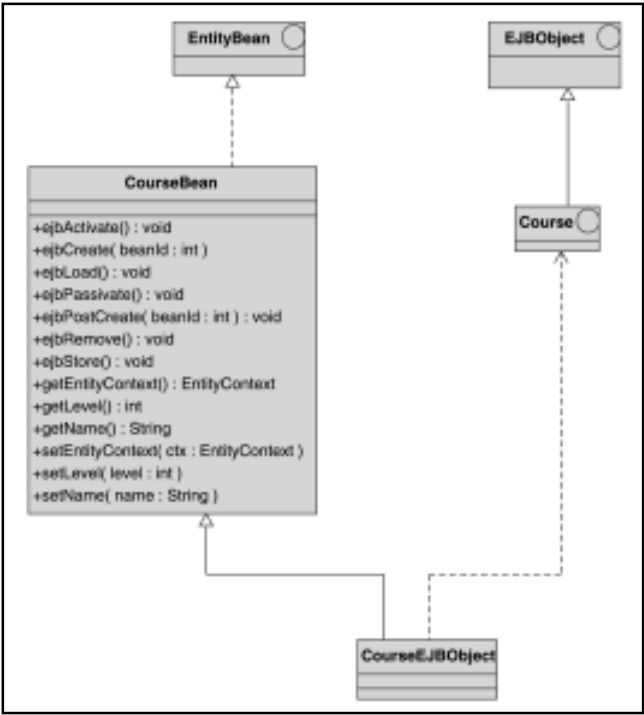
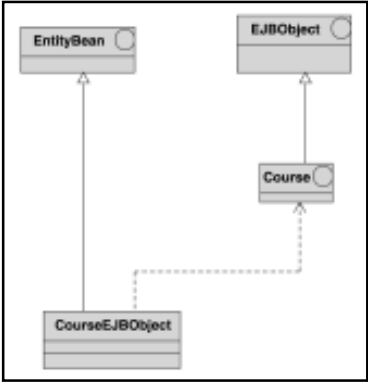



Figura 17.6 – *Il* `CourseBean` *non implementa la classe* `CourseBean`*, ma la ingloba al suo interno secondo un modello proprietario del server*

Secondo il pattern Delegation, il client, utilizzando lo schema tipico di RMI, può utilizzare tale stub per invocare i metodi dell'EJBObject, il quale inoltrerà tali chiamate al bean stesso.

Utilizzare gli EJB

Si può passare a questo punto ad analizzare quali siano i passi necessari per utilizzare un bean da parte di una applicazione client.

Per prima cosa è necessario ricavare un reference alla home interface, in modo da poter ricavare i vari stub degli oggetti remoti. Maggiori approfondimenti verranno dati in seguito, per il momento basti sapere che si utilizza per questo scopo la API Java Naming and Directory Interface (JNDI)

```
CourseHome = ... // tramite JNDI
```

A questo punto, dall'oggetto remoto si utilizza una chiave univoca dell'oggetto remoto che si desidera utilizzare per ricavare l'oggetto stesso.

Tale informazione, in accordo alla definizione data della classe `CoursePK`, potrebbe essere memorizzata in una variabile di tipo intero, ad esempio

```
int PrimaryKey = 47;
```

Il reperimento del reference remoto può essere fatto attraverso poche righe di codice

```
CoursePK pk = new CoursePK(PrimaryKey);
Course course = CourseHome.findByPrimaryKey(pk);
System.out.println("Oggetto remoto ricavato" + course.getName());
```

Da questo breve esempio si può intuire come in realtà l'utilizzazione di un bean si traduca nella sua istanziazione da parte del client (questo è il passaggio più importante, ed è stato volutamente tralasciato dato che verrà affrontato dettagliatamente in seguito), e nel successivo utilizzo tramite invocazione dei metodi pubblici.

Lo scopo di un entity è quello di rappresentare un oggetto da utilizzare contemporaneamente da parte di più client: ogni entity bean è pensato per mantenere sincronizzati i dati fra i vari client, in modo da centralizzare le informazioni. Questo comporta una notevole riduzione della ridondanza delle risorse utilizzate dai vari client.

Gli entity bean rappresentano quindi un ottimo modo per modellare strutture dati secondo il paradigma a oggetti, ma non sono indicati per esprimere un'operazione o un determinato compito.

I session beans invece operano come agenti al servizio del client, effettuando tutte quelle operazioni che coinvolgono lo scambio di informazioni fra entity beans, e l'esecuzione di computazioni particolari sia fini a se stesse, sia con effetto su altri entity beans, dando vita al workflow della applicazione.

Qui di seguito è riportato un esempio che mostra come un client possa sfruttare i servizi di un session bean e di altri entity. Ad esempio per effettuare una prenotazione di un corso e il successivo pagamento, si potrebbe scrivere

```
String CreditCard = "112233445566";
String CourseName = "Introduzione a Enterprise Java Beans";

// Alumn è un entity bean creato precedentemente
CourseAgent ca = CourseAgentHome.create(Alumn);
ca.setCourseName(CourseName);

badge = ca.enterForCourse(CreditCard, Alumn);
```

In questo caso tutte le operazioni sono effettuate in maniera trasparente dal session bean `CourseAgent` il quale, oltre a controllare i dati immessi, effettua il pagamento utilizzando il numero di carta di credito passato come parametro.

Il pagamento verrà effettuato per mezzo di un altro session bean (vedi oltre) il cui compito è quello di effettuare i necessari controlli e di accreditare la cifra indicata su un determinato conto corrente.

Dato che si opera in un panorama distribuito, ogni operazione, invocazione di metodi o passaggio di parametri potrebbe comportare un inutile traffico di rete: spostando invece tutte le operazioni sul server si limita lo scambio di dati serializzati fra client e server.

Discorso analogo si può fare anche per il numero di connessioni aperte fra client e server.

L'habitat dei Bean

L'ambiente all'interno del quale un bean vive e opera è composto da due oggetti, il container e il server. Se mentre il primo è deputato al controllo del ciclo di vita del bean, il container rappresenta invece un concetto piuttosto vago, utilizzato esclusivamente per spiegare il funzionamento di EJB e per definirne il modello.

Definire un container equivale a specificare il comportamento e la modalità di interfacciamento (la cui base fondamentale sono le interfacce `EntityBean` e `SessionBean`) verso i vari oggetti remoti, specifica che deve essere adottata fedelmente dai vari costruttori di server EJB.

È libera invece l'implementazione della parte relativa al server, per permettere la massima libertà progettuale e implementativa ai vari costruttori di creare il loro prodotto EJB compliant.

Di fatto questo ha anche il grosso vantaggio di permettere la creazione di server EJB da distribuire come parte integrante di application server già presenti sul mercato ma non espressamente pensati in origine per supportare il modello EJB (vedi il caso di IBM WebSphere o BAS di Borland).

I servizi di sistema

A partir dalla gestione delle varie istanze di bean, fino al controllo sulle transazioni, i servizi di sistema rappresentano probabilmente il punto più importante di tutta la tecnologia EJB, dato che permettono la cosiddetta separazione fra il lavoro del bean developer e quello del server.

Ad esempio senza la gestione dei pool di bean, si dovrebbe implementare manualmente tutta una serie di importanti aspetti, lavoro questo sicuramente lungo e insidioso, tale da rendere inutile l'utilizzo di uno strumento potente ma anche complesso come lo sono gli application server per EJB. Senza il supporto per i servizi di base EJB perderebbe tutta la sua importanza e non offrirebbe nessun motivo di interesse specie al cospetto di alternative come CORBA o RMI.

Nel seguito di questa sezione si vedranno più o meno in profondità gli aspetti principali legati alla gestione dei servizi più importanti, cercando di dare una giustificazione del loro utilizzo e del loro funzionamento. In alcuni casi, come per esempio per le transazioni, verrà fatto un ulteriore approfondimento.

Resource Management

Uno dei maggiori vantaggi offerto dal Component Transaction Model (CTM) è rappresentato dall'elevato livello di prestazioni raggiungibile in rapporto al carico di lavoro, ovvero in funzione del numero di client che accedono ai vari oggetti remoti messi a disposizione dal server.

Per questo motivo il primo aspetto che si andrà a considerare è quello relativo alla gestione delle risorse condivise, ovvero dei client che accedono ai bean installati sul server.

È bene tener presente che, come in tutti gli scenari concorrenti, specie nel caso di internet, il numero dei client in genere non è prevedibile. È quindi ovvio che tanto migliore sarà l'ottimizzazione della gestione e degli oggetti in esecuzione contemporanea, tanto migliori saranno le prestazioni complessive del sistema.

Il meccanismo utilizzato in questo caso, preso in prestito dai sistemi di gestione di basi di dati, è quello del pooling delle risorse.

Alla base di tale architettura vi è la considerazione secondo la quale molto raramente si verifica l'ipotesi per cui tutti i client dovranno accedere nello stesso istante ai vari bean installati sul server: di rado si renderà quindi necessario istanziare e referenziare contemporaneamente tutti gli oggetti remoti che corrispondono ai molti client in funzione.

Oltre a una drastica riduzione del numero di connessioni aperte e degli oggetti attivi, il meccanismo del pool di oggetti remoti risulta essere anche più efficiente: infatti come si avrà modo di vedere più avanti, il meccanismo di condivisione risulta più efficiente rispetto alla creazione/distruzione sia di connessioni che di bean. In definitiva solo pochi oggetti remoti verranno realmente istanziati ed utilizzati, dando vita ad una politica di condivisione dei bean tramite le molte interfacce istanziate.

Questa separazione fra ciò che è in funzione sul lato server, e quello che invece il client gestisce è reso possibile grazie al fatto che il client non accede mai direttamente agli oggetti remoti, ma sempre tramite interfacce remote, secondo lo standard di EJB.

Pooling di Entity Beans

Per comprendere come sia messo in atto il sistema di pooling, si prenda prima in esame il caso degli entity bean. Tali componenti vivono il loro ciclo di vita passando dei seguenti stati:

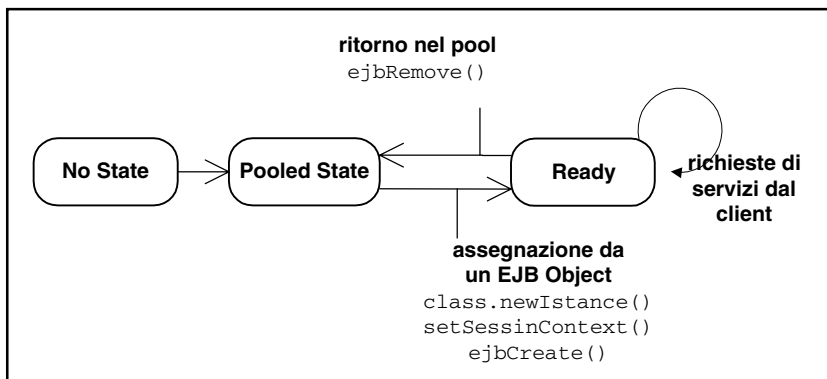
- *No state*: è lo stato iniziale del bean, quando non è stato ancora instanziato e quando nessuna risorsa è stata ancora allocata per quel componente.

- *Pooled state*: il componente è stato istanziato dal container, ma ancora non ha nessun EJB Object associato.

- *Ready State*: infine il bean può essere ready, ovvero è stato istanziato ed associato ad un EJB Object.

Sulla base di tale organizzazione risulta intuitivo come sia gestito il pooling: il server infatti istanzia una serie di componenti remoti, ponendoli nel pool, ed associando un componente remoto ad un determinato EJBObject solo al momento dell'effettivo bisogno.

Tutti i componenti remoti sono quindi equivalenti fra loro mentre sono nello stato pooled, ed acquistano una contestualizzazione solo al momento della invocazione da par-

Figura 17.7 – *Ciclo di vista di un componente EJB nell'ambito del server*

te del client di uno dei business methods del bean: il client effettua le invocazioni ai metodi del wrapper (EJBObject) e non direttamente dell'oggetto remoto.

Nella fase di *ready* il componente riceve le invocazione in callback dal server e non dal client.

Appena un componente viene istanziato e posizionato nel pool, riceve un nuovo context (istanza di un `javax.ejb.EJBContext`) che offre una interfaccia al bean per comunicare con l'EJB environment.

L'`EJBContext` è in grado sia di reperire informazioni sul client che ha effettuato le invocazioni, sia di fornire un riferimento alle interfacce `EJBHome` ed `EJBObject` in modo da permettere l'invocazione da parte del bean stesso dei metodi di business di altri componenti.

Quando il bean remoto viene invalidato, nel caso in cui il client invochi uno dei metodi di rimozione, oppure perché il componente è uscito dallo scope di esecuzione, il bean viene separato dall' `EJBObject` e successivamente riposizionato nel pool.

Può anche accadere che un bean sia reimmesso nel pool dopo un periodo di inutilizzo prolungato da parte del client: in tal caso, nel momento in cui il client dovesse richiedere nuovamente l'utilizzo del bean, allora verrà effettuata nuovamente una operazione di assegnazione prelevando dal pool il primo componente disponibile.

Questa operazione detta instance swapping è particolarmente importante dato che permette di servire un alto numero di richieste utilizzando poche risorse attive: il tempo in cui il bean viene utilizzato dal client statisticamente è mediamente minore del tempo passato in attesa nel pool. Essendo l' instance swapping è una operazione meno costosa della creazione di un bean, giustifica quindi l'utilizzo di un pool di bean.

Pooling di Session Beans

Nel caso degli stateless il meccanismo di pool è semplificato dalla natura stessa del componente: in questo caso infatti non viene mantenuta nessuna informazione fra due invocazioni dei metodi da parte del client, e quindi non è necessario memorizzarne lo stato. Ogni metodo esegue le istruzioni senza che si debba accedere in qualche modo ad informazioni memorizzate da qualche parte.

Questo permette al container di effettuare il processo di swapping senza tener conto di quale particolare istanza venga utilizzata.

Nel caso degli stateful session beans invece le cose sono leggermente differenti: l'integrità delle informazioni deve essere mantenuta in modo da ricostruire in ogni momento la storia delle varie invocazioni (che si definisce *conversational state*, ovvero stato della conversazione fra client ed bean); per gli stateful beans quindi non viene utilizzato il meccanismo di swapping di contesto.

In questo caso tutte le volte in cui sia necessario utilizzare un componente, è sufficiente prelevarne uno vuoto dalla memoria (quindi in modo simile al sistema di pool), ripristi-

nandone lo stato prelevando le informazioni direttamente da un sistema di memorizzazione secondaria (tipicamente file serializzati su file system). Questa operazione viene detta *attivazione*, ed è simmetrica a quella di scrittura su disco che prende il nome di *passivazione*.

Un aspetto piuttosto importante è quello legato agli eventuali campi definiti come transient: il meccanismo di salvataggio e ripristino dello stato si basa infatti sulla serializzazione, la quale, come noto, impone il ripristino ai valori di default per i campi transient nel momento della deserializzazione dell'oggetto: per gli interi ad esempio il valore 0, per i reference null, per i boolean false e così via.

In questo caso però non si può avere la certezza matematica dello stato dell'oggetto al suo ripristino, visto che la specifica EJB 1.0 lascia libertà di scelta all'implementatore del server. Per questo motivo, dato che l'utilizzo dei campi transient non è necessario tranne che in rare occasioni, se ne sconsiglia l'utilizzo.

Si ricorda che i metodi di callback utilizzati dal server per informare degli eventi di attivazione e passivazione sono i due `ejbActivate()` ed `ejbPassivate()`: il primo viene invocato immediatamente dopo l'attivazione, ed il secondo immediatamente prima la passivazione del componente.

Entrambi sono particolarmente utili in tutti quei casi in cui si desideri realizzare una gestione particolareggiata delle risorse esterne al server; ad esempio nel caso in cui bean debba utilizzare una connessione con un database o con un qualsiasi altro sistema di comunicazione.

Infine nel caso degli entity beans non si parla di mantenimento dello stato, ma piuttosto di persistenza delle informazioni memorizzate nel componente: i dati vengono in questo caso memorizzati in un database non appena una qualche modifica viene effettuata nel componente.

La gestione della concorrenza

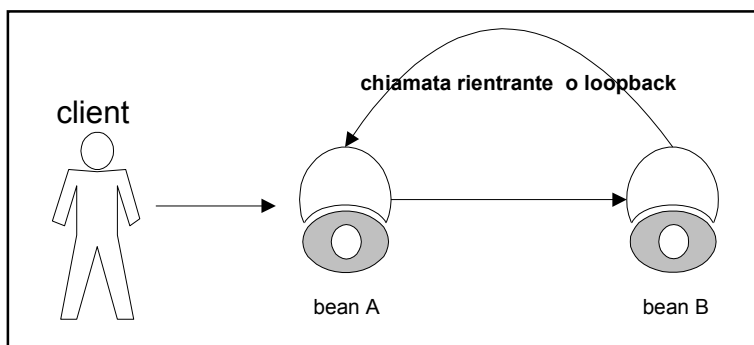
Quello della concorrenza è un problema molto importante, ma che per fortuna nel caso dei session beans non si pone, grazie alla natura stessa di questi componenti.

Ogni componente stateful infatti è associato ad uno ed un solo client: non ha senso quindi pensare ad uno scenario concorrente, essendo soggetto invocante sempre lo stesso.

Anche nel caso degli stateless l'accesso concorrente perde di significato, dato che questo tipo di oggetti non memorizzano nessun tipo di stato, e quindi mai due bean potranno interferire fra loro.

Nel caso degli entity invece il problema della concorrenza si pone in tutta la sua drammaticità: questo tipo di bean infatti rappresenta un dato non legato ad un particolare client, dando luogo ad un tipico scenario di accesso concorrente.

La soluzione adottata in questo caso è la più conservativa, limitando l'accesso contemporaneo non solo a livello di risorsa specifica, ma più precisamente di thread di esecuzione. Per thread di esecuzione si intende ad esempio una serie di soggetti in esecuzione conseguente (A invoca B che invoca C).

Figura 17.8 – *L'invocazione rientrante di un metodo può dar vita a problemi di deadlock*

Fra le molte conseguenze di una scelta così conservativa, forse la più immediata è quella relativa alla proibizione dello statement `synchronized`, in modo da lasciare al server la gestione degli accessi concorrenti.

Questa politica, pur limitando notevolmente l'insorgere di possibili problemi, lascia scoperto il caso delle cosiddette chiamate rientranti o loopback calls. Si immagini ad esempio il caso in cui un client invochi un metodo di un bean A: in questo caso il thread di esecuzione facente capo al client acquisisce il lock associato al componente A.

Nel caso in cui il metodo di B debba a sua volta invocare un metodo di A, si otterrà un blocco di tipo deadlock, dato che B attende lo sblocco del lock su A per poter proseguire, evento che non si potrà mai verificare.

Questo scenario nasce dalla scelta di voler attivare il lock a livello di thread di esecuzione, piuttosto che singole entità.

Figura 17.9 – *L'invocazione rientrante di un metodo può dar vita a problemi di deadlock*

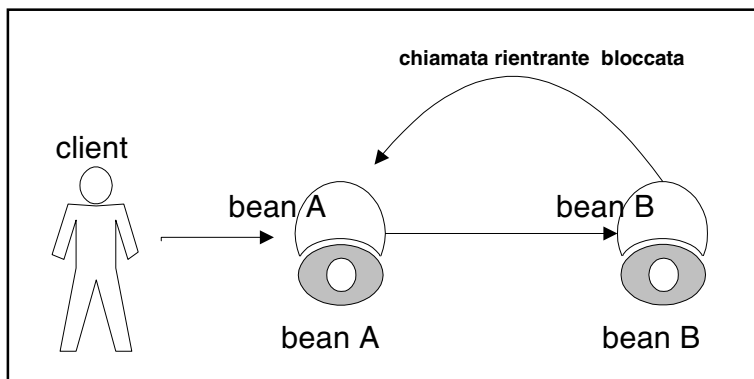
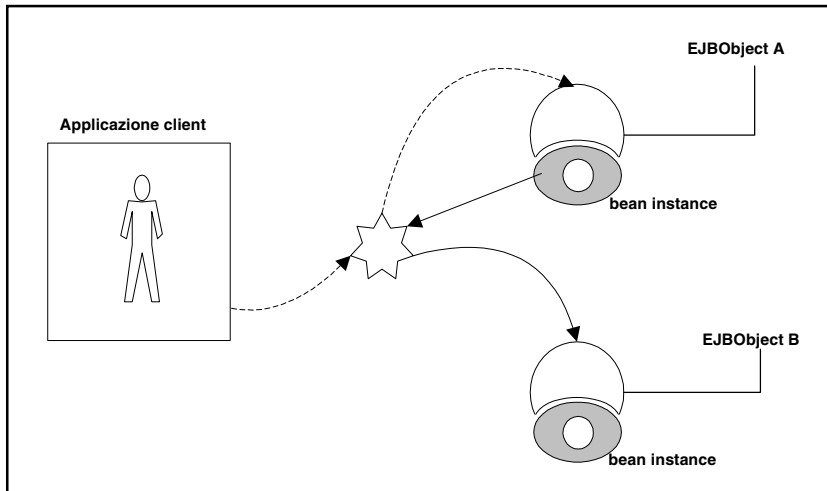



Figura 17.10 – *L'utilizzo di un componente, anche se locale, avviene sempre tramite il meccanismo dell'invocazione remota*

La soluzione a questo problema è quello di proibire il loopback: per i session la sua presenza porta ad una `RemoteException`, mentre gli entity possono essere configurati per funzionare anche in situazioni di loopback, anche se questa è una scelta fortemente scoraggiata.

Ovviamente un componente può accedere ai suoi metodi (ovvero un metodo di A può invocare senza problemi un altro metodo di A) senza dover sottostare ai var vincoli di lock, come del resto avviene nel caso della classe `Thread`.

Ogni invocazione di un metodo di un bean avviene sempre e comunque secondo lo schema classico della invocazione remota basata su `EJBObject`. Quindi nell'esempio appena visto, il client C invoca i metodi di A secondo lo schema visto: conseguentemente A diviene client di B, e quindi invoca i suoi metodi ricavandone l'interfaccia remota. B a sua volta nella chiamata di loopback ritorna ad essere client di A, e quindi, grazie al meccanismo di lock sui thread non può eseguire tale operazione.

Anche se questa organizzazione può apparire eccessivamente complessa, presenta due importanti vantaggi: da un lato permette di mantenere un elevato livello di standardizzazione, dato che ogni componente deve essere progettato ed implementato seguendo una rigida specifica standard. Dall'altro, visto che A invoca i metodi di B tramite una interfaccia remota, è possibile effettuare sostituzioni, e migrazioni del bean da un server ad un altro, senza che il client ne abbia percezione. Questo permette massima flessibilità, semplicità di gestione e con poco lavoro consente l'implementazione di tecniche di load balancing fra server differenti.

Passare per forza sempre dall'`EJBObject` rappresenta però anche il principale tallone d'Achille di tutta l'architettura EJB, dato che può provocare grossi rallentamenti in fase di esecuzione: ogni invocazione infatti è vista come una invocazione remota alla RMI anche se l'oggetto è in esecuzione localmente (come potrebbe essere nel caso di B).

Le transazioni

Il modello di EJB fornisce un robusto supporto per le transazioni. Anche in questo caso il programmatore del componente non deve implementare nessuna struttura particolare, dato che è sufficiente definire i dettagli operativi momento del deploy del componente: indicando, ad esempio tramite file XML, quali operazioni dovranno essere atomiche e specificando un contenitore di transazioni, si può fornire il supporto transazionale per un intero bean piuttosto che per i singoli metodi.

Vista l'importanza e la complessità dell'argomento, si rimanda l'approfondimento di questi aspetti al paragrafo relativo presente più avanti nel capitolo.

La gestione della persistenza

La gestione della persistenza delle informazioni è un aspetto molto importante, anche se riguarda esclusivamente il caso degli entity bean. Questi oggetti di fatti rappresentano dati o collezione di dati che possono essere memorizzati tramite l'utilizzo di un database di vario tipo. Tale sincronizzazione con la memoria secondaria permette fra le altre cose, di realizzare applicazioni robuste e fault tolerant: ad esempio in seguito ad un blocco del sistema, ogni componente potrà riassumere la sua originaria configurazione rileggendo i dati memorizzati nel database.

La gestione della persistenza può essere effettuata in due modi: container managed o bean managed (vedi oltre).

Nel primo caso è il container che si preoccupa di effettuare il mapping e la sincronizzazione fra i dati contenuti nel componente, e quelli memorizzati nel database. Nel secondo caso invece, sempre appoggiandosi ad un archivio esterno, tutta la logica di sincronizzazione dei dati deve essere implementata a mano.

Come presumibilmente appare ovvio la prima soluzione è quella più comoda e sicura, mentre la bean managed viene utilizzata ad esempio quando sia necessaria una personalizzazione del processo di salvataggio dei dati, oppure quando si debba comunicare con codice legaci.

In questo caso una soluzione basata su EJB risulta essere particolarmente importante ed efficace dato che permette di esportare una interfaccia Java di alto livello su un sistema legacy basato su tecnologie obsolete o comunque non standard. I vari componenti remoti funzioneranno come contenitori in comunicazione con la parte legacy ad esempio via socket o per mezzo di JNI.

Mapping dei dati e modello di memorizzazione

Il legame fra bean e dati memorizzati nel database è talmente stretto che una modifica nel primo si riflette in un aggiornamento dei dati: viceversa inserire a mano un nuovo record (ad esempio tramite uno statement SQL nel caso in cui si sia optato per un database relazionale) corrisponde ad creare una istanza logica di un nuovo bean, anche se non necessariamente tale operazione produce realmente nuove classi Java.

Il database da utilizzare per la memorizzazione di un entity può essere di tipo relazionale oppure ad oggetti.

Il modello relazionale è quello adottato nella maggior parte dei casi a causa delle maggiori prestazioni, affidabilità e maturità dei prodotti rispetto ad altre tecnologie, e soprattutto perché è più conosciuto dagli utilizzatori finali.

La grossa limitazione di questo modello è quello di essere estraneo alla logica ad oggetti utilizzata in EJB ed in Java in genere: si rende necessario quindi uno strato di software con cui effettuare il mapping relazionale-object oriented, in modo da trasformare i campi di un bean in valori delle colonne di un database.

Purtroppo non sempre è possibile effettuare in modo lineare ed automatico questo passaggio, tanto che in molti casi si rendono necessarie conversioni manuali, come ad esempio nel caso in cui particolari oggetti Java non siano direttamente convertibili nel formato accettato dal database. Inoltre la struttura relazionale può complicarsi molto in funzione alla reale struttura del bean di partenza.

I database ad oggetti invece non richiedono nessuna operazione di conversione, e quindi permettono ad un componente di essere salvato in modo diretto nel database stesso, consentendo di limitare la complessità della struttura.

Lo svantaggio in questo caso risiede nella immaturità dei vari database ad oggetti, nelle prestazioni non paragonabili con il modello relazionale, e nella mancanza di una diffusa cultura di tali strumenti fra gli sviluppatori ed utilizzatori.

Servizio di Naming

Qualsiasi sistema di programmazione distribuita ha alla base un qualche meccanismo che permetta di rintracciare oggetti remoti in esecuzione in spazi di indirizzamento eterogenei. Un sistema di questo tipo si basa sui cosiddetti naming services che sono essenzialmente di due tipi: servizio di bind (ovvero registrazione di un oggetto remoto in un qualche registry tramite nome logico) e servizio di lookup (che corrisponde alla operazione inversa, ovvero ricercare la particolare istanza di un oggetto remoto, partendo da un nome logico).

Nel caso di EJB il meccanismo utilizzato è quello offerto dalla Java Naming and Directory Interface (JNDI API). Questo strumento implementa una astrazione gerarchica ad alto livello di una ipotetica collezione di risorse (file, oggetti, stampanti, devices vari). Per una trattazione approfondita del funzionamento di JNDI si può far riferimento alla bibliografia, e in particolare [JNDI].

JNDI non rappresenta l'unico strumento di naming utilizzabile all'interno di un server EJB, anche se la specifica dice che ogni produttore di server deve fornire almeno una interfaccia di questo tipo ai vari client che intendano utilizzare i bean installati.

Da questo punto di vista il client deve utilizzare la JNDI API per iniziare la connessione verso un EJB Server specificando l'interfaccia `EJBHome` dell'oggetto remoto che si intende utilizzare. Ad esempio si potrebbe scrivere

```
import javax.naming.*;
...
Context jndiContext = new  InitialContext(props);
MyHome home = (MyHome) jndiContext.lookup(beanName);
MyBean  myBean =Home.create(param1, param2);
myBean.doSomething();
```

In questo caso `props`, istanza di `Properties`, indica al servizio JNDI dove si trova il server EJB da utilizzare; `beanName` invece indica il nome logico con cui il componente è stato registrato.

Security

La gestione della sicurezza è uno degli aspetti chiave su cui si basano la maggior parte dei moderni sistemi di comunicazione e di programmazione distribuita.

Si può intendere la sicurezza a tre livelli: autenticazione, access control e sicurezza nella comunicazione.

Nel primo caso, grazie ad un qualche sistema di riconoscimento si deve consentire a un utente (persona fisica) o entità (ad esempio un oggetto) l'accesso al sistema e di poter usufruire dei servizi messi a disposizione. Questo sistema in genere è considerato non sufficientemente flessibile, dato che permette di attivare o disattivare del tutto il set delle possibili operazioni effettuabili. Per questo spesso si parla di meccanismo On/Off.

Il riconoscimento può avvenire tramite password, certificato digitale o tramite meccanismi particolari come smart card o devices elettromagnetici.

Anche se in EJB non è formalmente specificato nessun meccanismo per permettere l'autenticazione, grazie ai servizi offerti da JNDI, si può comunque implementare un qualche sistema di riconoscimento: ad esempio si potrebbero utilizzare le informazioni per il login al momento in cui si effettua l'operazione di lookup semplicemente passando i dati utente come proprietà al context che effettua la ricerca; di seguito ecco un breve esempio

```
import javax.naming.*;
...
Properties props = new Properties();
props.put(Context.SECURITY_PRINCIPAL, uid);
props.put(Context.SECURITY_CREDENTIALS, passwd);
```

```
Context jndiContext = new  InitialContext(props);
...
```

Nel caso si possa disporre di un sistema di access control, allora è possibile definire una policy di permessi, in modo da autorizzare un utente ad effettuare determinate operazioni. Questa tipologia di security è quella incluso nella specifica 1.0 di EJB.

In questo caso ad ogni client viene associato un elemento di istanza di `java.security.Identity` che rappresenta l'identità con cui il client potrà effettuare le operazioni con gli oggetti remoti. Una `Identity` rappresenta un utente o un ruolo nel caso in cui l'oggetto remoto sia invocato da un altro bean, che come specificato in precedenza assume il ruolo di client. Questa puntualizzazione è molto importante, dato che colma una carenza presente molto spesso in sistemi distribuiti: ad esempio ci si potrebbe chiedere nel caso di un servlet, con quali diritti esso sia mandato in esecuzione (generalmente con quelli dell'utente che ha fatto partire il servlet container, ma non è detto).

La `Identity` associata ad un client viene utilizzata in modo del tutto trasparente: ad esempio quando il client richiede l'utilizzo di un particolare metodo, alla invocazione dello stesso viene passata anche l'istanza della identità associata, al fine di effettuare il controllo sui diritti. La definizione delle varie policy si effettua al momento del deploy grazie al `DeploymentDescriptor` il quale contiene i varie istanze di `ControlDescriptor` ed `AccessControlEntry`: questi ultimi permettono di definire la lista degli utenti ammessi ad effettuare determinate azioni, mentre i `ControlDescriptor` consentono fra l'altro di specificare il "runAs" di ogni `Identity`, ovvero con quale identità ogni metodo potrà essere eseguito.

Ad esempio si può specificare che un metodo qualsiasi di un determinato componente possa essere eseguito solo dall'utente "pippo", ma che tale metodo poi verrà eseguito nel sistema come se si trattasse dell'utente "amministratore di sistema". Si intuisce quindi come tutti questi meccanismi possano garantire un altissimo livello di personalizzazione del sistema in fatto di sicurezza. Il prezzo da pagare è ovviamente la complessità del tutto, come già in passato è avvenuto con l'introduzione dei sistemi avanzati di crittografia in Java, o con l'avvento del processo di firma digitale delle applet.

Infine si può intendere la sicurezza come quel meccanismo atto a rendere le comunicazioni non decifrabili da estranei (in EJB l'invocazione dei metodi ed i parametri passati). Quasi sempre questo si traduce nel dover utilizzare sistemi di crittografia basati su protocolli particolari. Al momento nella specifica EJB non è stata formalmente definita nessuna tecnica per supportare meccanismi di questo tipo.

Gli Entity Beans

Lo scopo principale di un entity bean è quello di memorizzare delle informazioni ed offrire al contempo una serie di metodi per la gestione da remoto di tali dati.

In quest'ottica è possibile quindi suddividere le funzionalità (metodi) di un bean in due categorie: da una parte si trova tutto ciò che è relativo alla sua gestione da remoto tramite il client, ovvero la cosiddetta business logic, mentre dall'altro si trova ciò che è relativo alla gestione del ciclo di vita del componente, del mantenimento dello stato e della gestione della sicurezza.

Uno degli aspetti più importanti degli entity bean è come questi implementano la persistenza dei dati all'interno del database. Due sono le possibilità: da un lato tutto il processo di sincronizzazione viene gestito dal container in modo trasparente, mentre nell'altro caso tutte le informazioni sono salvate su disco direttamente tramite operazioni effettuate dal bean. Nel primo caso si parla di bean Container Managed Persistence (CMP), mentre nell'altro si utilizza la definizione Bean Managed Persistence (BMP). Per il momento verrà effettuata una breve introduzione a tali concetti, rimandando alla paragrafo dedicato alla parte pratica per una analisi più dettagliata.

In un bean di tipo CMP il server si preoccupa in modo del tutto trasparente di sincronizzare le informazioni contenute nel bean al variare dello stato e soprattutto in funzione dello stato assunto all'interno del ciclo di vita del componente.

Tranne per il caso delle variabili definite transient, tutti gli oggetti serializzabili, o le variabili primitive possono essere salvate. Dato che nella maggior parte dei casi il database utilizzato è di tipo relazionale un utilizzo di variabili primitive sicuramente semplifica questo passaggio.

Gli entity bean di tipo CMP possono risultare particolarmente utili nel caso in cui sia semplice effettuare un mapping bean-database: quando il processo di persistenza del componente richiede invece un trattamento particolare, sia per la maggiore complessità del caso, sia per la necessità di utilizzare tecniche di persistenza particolari, si può passare ad implementare una gestione manuale della sincronizzazione fra componenti e dati nel database.

Il prezzo da pagare in questo caso è dato dalla maggiore complessità de codice del bean, anche se il passaggio da CMP ad un BMP è relativamente semplice grazie alla filosofia di base del modello EJB: il programmatore infatti dovrà semplicemente implementare i vari metodi di callback invocati dal server (come `ejbCreate()` o come quelli di sincronizzazione `ejbStore()` ed `ejbLoad()` e quelli di ricerca) al fine di definire "cosa" deve essere fatto in corrispondenza di un determinato evento; il "quando" questo deve essere fatto è demandato al server che gestisce il bean ed il suo ciclo di vita.

Di seguito sono riportate le considerazioni più importanti relative ad ognuno di questi metodi. Il metodo

```
ejbCreate()
```

viene invocato indirettamente dal client quando questo invoca il metodo create() della interfaccia home, ovvero quando genera un nuovo componente. Nel caso dei BMP questo metodo è responsabile di creare i dati relativi al bean all'interno del database. Oltre alla

diversa implementazione del corpo del metodo, rispetto ai CMP la differenza principale è il diverso valore ritornato: in questo caso infatti è obbligatorio che sia ritornata la chiave primaria del nuovo entity, e non `void` (null in EJB 1.0) come per i bean CMP.
Da un punto di vista strettamente implementativo, nel caso in cui il database d'appoggio scelto sia di tipo relazionale, all'interno del metodo troveremo delle istruzioni insert SQL, che effettueranno una o più insert delle varie variabili d'istanza del componente. Si ricordi che tutte le eccezioni SQL dovranno essere wrappate e propagate come `EJBExceptions` o `RemoteExceptions`. Il metodo `ejbPostCreate()` invece, non è coinvolto dal processo di persistenza utilizzata e deve sempre ritornare void.

`ejbLoad()`
`ejbStore()`

Questi due metodi sono invocati tutte le volte che il container decide che sia necessaria una sincronizzazione fra i dati memorizzati nel database, e le variabili del componente. Il primo verrà invocato in concomitanza di una transazione o prima di un business methods, in modo da avere la certezza che il bean contenga i dati più recenti contenuti nel database.

Il corpo del metodo non sarà molto differente da `ejbCreate()`, tranne che in questo caso, sempre nell'ipotesi di utilizzare un database relazionale, si troveranno delle update SQL piuttosto che delle insert. Continuano a valere anche in questo caso le considerazioni sulla gestione wrappata delle eccezioni.

`ejbRemove()`

Considerazioni analoghe a quelle dei casi precedenti sono quelle relative al metodo `ejbRemove()`, dove di fatto si deve provvedere alla cancellazione dei dati nel database.

Metodi di ricerca: per ogni metodo di ricerca presente nella home interface, dovrà essere presente un analogo nel bean. Si ricordi infatti che il client effettua le invocazioni direttamente sulla home che poi inoltra tali chiamate direttamente al bean. Nel caso dei BMP, i metodi di ricerca sono responsabili di trovare nel database i vari record che corrispondono ai criteri di ricerca impostati.

Ad esempio se nella home è presente codice del tipo

```
public interface MyHome extends javax.ejb.EJBHome{
   public MyBean findByPrimaryKey(MyBeanPK pk) throws FinderException,
                                                      RemoteException;
   public Enumeration findByName(String name)  throws FinderException,
                                                      RemoteException;
}
```

allora nel bean si dovrà scrivere

```
public class MyBean extends javax.ejb.EntityBean{
   public MyBeanPK ejbFindByPrimaryKey(MyBeanPK pk) throws FinderException,
                                                          RemoteException{}
    public Enumration ejbFindByName(String name) throws FinderException,
                                                          RemoteException{}
}
```

dove in questo caso `FinderException` è stata definita appositamente per segnalare l'insorgere di problemi durante la fase di ricerca.

Questi metodi appena visti sono quelli tramite i quali il server avverte il bean di un passaggio di stato o di cambiamento all'interno del ciclo di vita. La loro definizione è obbligatoria per correttezza, ma non è necessario implementarne un qualche comportamento. Un bean di tipo CMP infatti potrebbe non aver nessuna necessità di interferire con il ciclo di vita, demandandone al server la completa gestione. A prima vista l'implementazione o meno di tali metodi è quindi la principale differenza fra un CMP ed un BMP.

Dal punto di vista del client invece e di tutto il resto dell'applicazione, non vi è nessuna differenza fra utilizzare uno o l'altro tipo di bean.

La specifica EJB 1.1 permette di rendere persistenti anche campi di un bean che contengano riferimenti ad altri bean: in questo caso il costruttore del server deve implementare alcune tecniche piuttosto complesse al fine di effettuare il mapping più adatto possibile. Normalmente questo si traduce nella memorizzazione della chiave primaria (oggetto `PrimariKey`), degli oggetti `Handle` o `HomeHandle`, o di altri riferimenti che identifichino univocamente il bean contenuto.

La gestione della persistenza a carico del container semplifica molto il lavoro di implementazione del bean, ma complica non poco la fase di progettazione del server; infatti in questo caso le tecniche di persistenza devono essere il più generiche e automatiche possibili, visto che non si può fare nessuna assunzione a priori su come sia fatto il componente.

Nel caso invece dei bean-managed si possono implementare tecniche ad hoc, a seconda del caso. Come si potrà vedere in seguito, tale soluzione è concettualmente molto semplice, dato che si riconduce alla scrittura di porzioni di codice JDBC all'interno dei vari metodi invocati in modalità callback durante il ciclo di vita del bean.

Identificazione di un bean: la Primary Key

Quando un client effettua una operazione di lookup su un oggetto remoto, si deve poterlo identificare un maniera univoca fra tutti quelli messi a disposizione dal server. Per questo motivo ad ogni bean è associata una chiave, la `PrimaryKey`, che può essere costituita da una classe apposita, oppure da un campo del bean stesso. Nel primo caso la classe

generata ha un nome che segue lo schema `NomeBeanPK`, e contiene al suo interno tutte le informazioni necessarie per individuare il bean. Ad esempio supponendo di avere un bean denominato `MyBean`, si potrebbe scrivere

```
public void MyBeanPK{
   public int beanId;
   public MyBeanPK(){}
   public MyBeanPK(int id){
   beanId=id;


   public boolean equals(Object obj){
      if (obj == null || !(obj instanceof MyBeanPK))
         return false;
      else {
         if(((MyBeanPk)obj).beanId == this.beanId)
            return true;
         else
            return false;
      }
   }


   public int hashCode(){
      return this.beanId
   }

   // converte  in stringa il valore della chiave
   public String toString(){
      return this.beanId + "";
   }
}
```

In questo caso questa classe funge da wrapper per una variabile intera che rappresenta la chiave del bean. Per questo motivo sono stati ridefiniti i metodi `equals()` ed `hashCode()` in modo da operare su tale variabile: nel primo caso viene fatto un controllo con la chiave dell'oggetto passato, mentre nel secondo si forza la restituzione di tale intero (cosa piuttosto comoda ad esempio nel caso in cui si vogliano memorizzare i bean in una tabella hash, per cui due codici hash potrebbero essere differenti anche se fanno riferimento allo stesso id).

Si noti la presenza del costruttore vuoto, necessario per poter permettere al container di istanziare un nuovo componente in modo automatico e di popolarne gli attributi con valori prelevati direttamente dal database utilizzato per la persistenza. Questa operazione effettuata in automatico, viene eseguita piuttosto frequentemente durante il ciclo di vita del componente, ed è alla base della gestione della persistenza ad opera del container.

Per questo motivo tutti i campi della `PrimaryKey` devono essere pubblici, in modo che tramite la reflection il container possa accedervi in modo automatico. Alcuni server permettono di accedere, tramite tecniche alternative, anche a variabili con differente livello di accesso, ma questo diminuisce la portabilità della chiave.

Inoltre una `PrimaryKey` deve sottostare alle regole imposte per l'invocazione remota basata su RMI: questo significa che possono essere chiavi quelle classi serializzabili o remote in cui siano stati ridefiniti i metodi `equals()` ed `hashCode()`.

In alternativa all'utilizzo di `PrimaryKey` è possibile utilizzare direttamente dentro il bean campi di tipo `String` o wrapper di tipi primitivi (ad esempio `Integer`); in questo caso si parla di chiave semplice. Il metodo di ricerca in questo caso agisce direttamente sul campo del bean.

In tale situazione la chiave primaria non può essere un tipo primitivo, ma occorre utilizzare sempre un wrapper, al fine di garantire una maggiore standardizzazione della interfaccia: ad esempio il metodo `getPrimaryKey()` restituisce una istanza di `Object`, che verrà convertito a seconda del caso.

L'utilizzo di una classe apposita come chiave è da preferirsi se si desidera consentire ricerche basate su chiavi composte (uno o più campi), oppure quando sia necessaria una maggiore flessibilità.

La specifica EJB 1.0 lasciava indefinito questo aspetto, rimandando al costruttore del server la scelta di supportare o meno le chiavi semplici.

Con la versione 1.1 invece la scelta di quale campo utilizzare non viene fatta durante la progettazione del componente, ma piuttosto durante la fase di deploy grazie ad un apposito documento XML. Ad esempio si potrebbe decidere di scegliere un campo `beanId` contenuto nel bean stesso tramite la seguente sequenza di script XML

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <primary-field>beanId</primary-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Il poter definire in fase di deploy quale sia la chiave da utilizzare ha una importante impatto sulla portabilità: in EJB 1.0 infatti il progettista deve decidere a priori quale sarà il campo da utilizzare per tale scopo, dovendo quindi fare una serie di assunzioni su quello che sarà il database utilizzato per la persistenza (relazionale o ad oggetti) e sulla struttura dati da esso offerta (ad esempio struttura delle tabelle).

Rimandando alla fase di deploy tale scelta di fatto permette di svincolare completamente la fase di progettazione ed utilizzazione del componente dalla quella di utilizzo finale.

È per questo motivo quindi che tutti i metodi che hanno a che fare con la chiave lavorano con parametri di tipo `Object`, rimanendo così slegati dalla particolare scelta sul campo scelto come chiave.

Si potrebbe dire che questa impostazione facilita la creazione di un mercato di componenti terze parti, indipendenti dalla piattaforma d'utilizzo e pronti per l'uso.

I metodi di ricerca

La ricerca di un bean, una volta definite le chiavi, avviene tramite i metodi di ricerca messi a disposizione dalla interfaccia remota. La specifica infatti dice che tale interfaccia deve fornire zero o più metodi creazionali, ed uno o più metodi di ricerca. Nel caso dei container managed, i metodi di ricerca sono implementati automaticamente al momento del deploy, in base alla politica particolare implementata dal container per effettuare la ricerca, ed in base a quanto specificato nei parametri di deploy. In alcuni casi il server permette di specificare attraverso il tool di deploy anche il comportamento di tali metodi.

Il `findByPrimaryKey()` è il metodo standard, ovvero quello che consente la ricerca in base alla chiave primaria, ma non è detto che sia il solo.

Ad esempio potrebbe essere comodo effettuare ricerche sulla base di particolari proprietà del bean: in questo caso i nomi dei metodi seguono lo schema `findBy`*`NomeCampo`*`()`.

Nel caso di ricerche per campi differenti dalla chiave primaria si possono avere risultati multipli: per questo con la specifica 1.1 il risultato della ricerca può essere un oggetto di tipo `Collection`.

I metodi di ricerca che ritornano un solo oggetto devono generare una eccezione di tipo `FinderException` in caso di errore, mentre una `ObjectNotFoundException` nel caso in cui non sia stato trovato niente.

La sincronizzazione con il database: metodi `ejbLoad()` ed `ejbStore()`

I metodi `ejbLoad()` ed `ejbStore()` possono essere utilizzati per effettuare tutte quelle operazioni di manipolazione dei dati relativamente al momento di persistenza del bean. Nei BMP questo significa che conterranno le istruzioni SQL per la scrittura e lettura dei dati. Anche nel caso di un CMP tali metodi possono essere d'aiuto, ad esempio nel caso in cui vi sia la necessità di effettuare conversioni o preparazioni del bean prima che questo sia reso persistente, oppure subito dopo la lettura dal database.

Si pensi al caso in cui uno dei campi del bean sia costituito da un oggetto non serializzabile: nel pezzo di codice che segue si è preso in esame il caso di una ipotetica classe `MyTable` non serializzabile.

```
public class MyBean extends EntityBean {
   public transient MyTable table;
   public String strTable;
```

```
    public void ejbLoad(){
        strTable=table.toString();
    }

    public void ejbStore(){
        table= new MyTable(strTable);
    }

    ...
}
```

In questo caso la variabile `MyTable` è indicata transient, in modo da impedirne la memorizzazione automatica nel database e potere salvare invece la rappresentazione testuale della tabella tramite l'oggetto `strTable`.

Senza entrare nei dettagli dell'implementazione della classe `MyTable`, sarà sufficiente dire che il costruttore accetterà come parametro una stringa contenente i valori riga-colonna che costituiscono la tabella, e che il metodo `toString()` effettuerà l'operazione opposta.

Anche se l'esempio è volutamente poco significativo da un punto di vista applicativo, mostra con quale semplicità sia possibile implementare uno strato di pre e post elaborazione relativamente all'operazione di lettura e scrittura su database.

L'interfaccia `EntityBean`

L'interfaccia `EntityBean` definisce una serie di metodi di callback atti alla gestione dello stato di un bean.

```
public interface EntityBean extends EnterpriseBean{
    public abstract void ejbActivate() throws RemoteExcetpion;
    public abstract void ejbPassivate() throws RemoteExcetpion;
    public abstract void ejbLoad() throws RemoteExcetpion;
    public abstract void ejbStore() throws RemoteExcetpion;
    public abstract void ejbRemove() throws RemoteExcetpion;
    public abstract void setEntityContext(EntityContext ec) throws RemoteExcetpion;
}
```

La maggior parte di tali metodi però non sono particolarmente utili nel caso in cui sia il container a dover gestire la persistenza dei dati memorizzati nel componente, fatta eccezione per ciò che riguarda la gestione del contesto.

L'interfaccia `EntityContext` fornisce tutte le informazioni utili durante il ciclo di vita di un bean, indipendentemente dalla politica adottata per il mantenimento dello stato, e rappresenta il tramite fra il bean ed il proprio container.

Il primo metodo invocato successivamente alla creazione della istanza del bean è il `setEntityContext()` il quale passa al componente una istanza del contesto in cui

agisce: esso viene invocato prima che il bean appena creato entri nel pool delle istanze pronte per l'utilizzo.

Ripensando al ciclo di vita di un bean, visto in precedenza, si potrà rammentare come in questa situazione le varie istanze non sono ancora state popolate con i dati prelevati dal database, rendendole di fatto tutte equivalenti. È al momento della invocazione da parte del client che un bean viene prelevato dal pool e contestualizzato: le sue proprietà vengono istanziate, con dati prelevati dal database, e l'istanza viene associata o "wrapperizzata" con un EJBObject per l'interfacciamento con il client.

In questa fase viene creato un contesto per il bean tramite la creazione e l'assegnazione di un oggetto `EntityContext`, e rappresenta il tramite fra il bean e l'EJBObject.

Il contesto inoltre è particolarmente importante se si riconsidera il meccanismo di pool e di swap di istanze dei vari bean: da un punto di vista implementativo infatti lo swapping è reso possibile grazie al continuo cambio di contesto associato al bean stesso il quale non necessita di sapere cosa stia succedendo fuori dal suo container o quale sia il suo stato corrente. Il bean in un certo senso "vive" il mondo esterno grazie al filtro del proprio contesto.

Fino a quando il metodo `ejbCreate()` non ha terminato la sua esecuzione, nessun contesto è assegnato al bean e quindi il bean non può accedere alle informazioni relative al suo contesto come la chiave primaria o dati relativi al client invocante: tali informazioni sono invece disponibili durante l'invocazione del metodo `ejbPostCreate()`.

Invece tutte le informazioni relative al contesto di esecuzione come ad esempio le variabili d'ambiente sono già disponibili durante la creazione.

Quando un bean termina il suo ciclo di vita, il contesto viene rilasciato grazie al metodo `unsetEntityContext()`.

Per quanto riguarda invece i metodi creazionali, benché acquistino particolare importanza nel caso di gestione della persistenza bean managed, possono rivelarsi utili per ottimizzare la gestione del componente e delle sue variabili.

Ogni invocazione di un metodo di questo tipo sulla Home Interface viene propagata sul bean in modo del tutto analogo a quanto avviene per i metodi di business invocati sulla remote Interface. Questo significa che si dovrà implementare un metodo `ejbCreate()` per ogni metodo corrispondente nella Home Interface.

La loro implementazione così come la loro firma dipende da quello che si desidera venga compiuto in fase di creazione. Tipicamente si tratta di una inizializzazione delle variabili di istanza.

Ad esempio si potrebbe avere

```
// versione EJB 1.1 che ritorna un bean
public MyBean ejbCreate(int id, String name, ...){
   this.beanId=id;
   this.beanName=name;
   ...
```

```
    return null;
}

// versione EJB 1.0  che ritorna un void
public void ejbCreate(int id, String name, ...){
    this.beanId=id;
    this.beanName=name;
    ...
}
```

Come si può notare la differenza fondamentale fra la specifica EJB 1.0 e 1.1 è il valore ritornato. Nella 1.0 infatti il metodo crea un componente senza ritornare niente, mentre nella 1.1 viene ritornato una istanza null della classe che si sta creando. Sebbene il risultato finale sia simile, dato che il valore ritornato viene ignorato in entrambi i casi, questa soluzione ha una importante motivazione di tipo progettuale: la scelta fatta nella specifica 1.1 permette infatti un più semplice subclassing del codice consentendo l'estensione di CMP da parte di un BMP. Nella versione precedente questo non era possibile dato che in Java non è permesso l'overload di un metodo che differisca solo per il tipo ritornato.

La nuova specifica permette quindi ai costruttori di server di supportare la persistenza container managed semplicemente estendendo un bean container managed con un bean generato di tipo bean managed.

La gestione del contesto

Il contesto in cui vive un bean, comune a tutti i tipi di componenti, è dato dalla interfaccia `EJBContext` dalla quale discendono direttamente sia la `EntityContext` che il `SessionContext`.

Ecco la definizione di tale interfaccia secondo la specifica 1.1

```
public interface EJBContext{

    public ejbHome getEJBHome();

    // metodi di security
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(String roleName);

    // metodi deprecati
    public java.security.Identity getCallerIdentity();
    public boolean isCallerInRole(Identity role);
    public Properties getEnvironment();

    // metodi transazionali
    public UserTransaction getUserTransaction(); throws IllegalStateException;
```

```
public boolean getRollbackOnly () throws IllegalStateException;
public boolean setRollbackOnly () throws IllegalStateException;
```

I metodi della `EJBContext` benché generici sono molto importanti: ad esempio `getEJBObject()` restituisce un reference remoto al bean object, per poter essere utilizzata dal client o da un altro bean.

I metodi della `EntityContext` e della `SessionContext` invece sono stati pensati per consentire una migliore gestione del ciclo di vita, della sicurezza e gestione delle transazioni di un entity o di un session.

Dal punto di vista del bean il contesto può essere utilizzato per avere un riferimento a se stesso tutte le volte che si deve effettuare una chiamata in loopback, ovvero quando un bean invoca un metodo di un altro bean passando come parametro un reference a se stesso: questa operazione infatti non è permessa tramite la keyword `this`, ma tramite un reference remoto ricavato dal contesto. I bean a tal proposito definiscono un metodo `getObject()` nella interfaccia `EntityContext`, il cui funzionamento è esattamente lo stesso. Ad esempio

```
public class MyBean extends EntityBean{
   public EntityContext context;

   public myMethod(){
      YourBean YB= ...
      EJBObject obj = Context.getObject() ;
      MyBean MB;
      MB = (MyBean)PortableObject.narrow(obj, MyBean.class);
      YB.yourMethod(MB);
   }
}
```

Il metodo `getEJBHome()`, disponibile sia per i session che per gli entity beans, è definito nella `EJBContext`: tale metodo restituisce un reference remoto al bean, e può essere utilizzato sia per creare nuovi bean, sia per effettuare delle ricerche nel caso di entity bean.

Durante il ciclo di vita all'interno del suo contesto, le varie informazioni accessibili tramite l'`EJBContext` possono variare, ed è per questo che tutti i metodi possono generare una `IllegaStateException`: ad esempio perde di significato ricorrere alla chiave primaria quando un bean si trova in stato di swapped, ovvero quando non è assegnato a nessun EJBObject, anche se può disporre di un `EJBContext`.

Il `getCallerPrincipal()`, che dalla versione 1.1 sostituisce il `getCallerIdentity()`, permette di ricavare il reference al client invocante.

Quando invece si deve implementare un controllo a grana più fine sulla sicurezza, il metodo `isCallerInRole()` permette di controllare in modo veloce ed affidabile il ruolo di esercizio del client invocante.

Per quanto riguarda i metodi transazionali invece, essi saranno affrontati in seguito, quando si parlerà in dettaglio della gestione delle transazioni.

Il deploy in 1.0 e 1.1

Dopo aver visto tutte le varie parti che compongono un entity bean, resta da vedere come sia possibile rendere funzionante tale componente, ovvero come effettuare il deploy di un EJB nel server. Le cose sono molto differenti a seconda che si desideri seguire la specifica 1.0 o 1.1: nel secondo caso le cose si sono semplificate moltissimo rispetto al passato, dato che è sufficiente scrivere in un documento XML tutte le informazioni necessarie. Molto spesso tale file viene generato dal tool di sviluppo: ad esempio JBuilder dalla versione 4 offre un editor visuale delle varie proprietà di funzionamento del bean, editor che poi rende possibile la creazione del file XML. Un esempio che si può ottenere con tale prodotto potrebbe essere il seguente

```
<?xml version="1.0" encoding="Cp1252"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1
//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

  <ejb-jar>
     <enterprise-beans>
        <session>
           <ejb-name>TellerBean</ejb-name>
           <home>quickstart.TellerHome</home>
           <remote>quickstart.Teller</remote>
           <ejb-class>quickstart.TellerBean</ejb-class>
           <session-type>Stateless</session-type>
           <transaction-type>Container</transaction-type>
        </session>
        <entity>
           <ejb-name>Accounts</ejb-name>
           <home>quickstart.AccountsHome</home>
           <remote>quickstart.Accounts</remote>
           <ejb-class>quickstart.AccountsBean</ejb-class>
           <persistence-type>Container</persistence-type>
           <prim-key-class>java.lang.String</prim-key-class>
           <reentrant>False</reentrant>
           <cmp-field>
              <field-name>name</field-name>
           </cmp-field>
           <cmp-field>
              <field-name>balance</field-name>
           </cmp-field>
           <primkey-field>name</primkey-field>
           <resource-ref>
```

```
            <res-ref-name>jdbc/accounts</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
      </entity>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>TellerBean</ejb-name>
            <method-name>*</method-name>
         </method>
         <method>
               <ejb-name>Accounts</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

Si noti come tramite appositi tag XML siano definite le varie caratteristiche sia dell'entity bean, sia del session e di come si sia potuto specificare il comportamento transazionale e la gestione della persistenza dei vari campi. L'esempio a cui si riferisce tale file XML è uno di quelli presenti nel tutorial Borland scaricabile direttamente dal sito della casa.

Per quanto riguarda invece il deploy di componenti secondo la specifica 1.0, le cose sono alquanto più complesse. In questo caso infatti il deploy descriptor è costituito da un oggetto Java serializzato, istanza di una classe che nel caso degli entity è la `EntityDescriptor`, che deriva dalla più generica `DeploymentDescriptor`. Nel caso dei session bean la classe da utilizzare sarebbe stata la `SessionDescriptor`.

Il codice necessario per gestire il deploy è generato anch'esso da appositi tool del server o dell'ambiente di sviluppo: dopo aver istanziato un oggetto tramite il costruttore

```
EntityDescriptor ed =  new  EntityDescriptor();
```

sarà possibile memorizzare tutte le informazioni relative al bean tramite i metodi messi a disposizione. Ad esempio si potrà indicare i nomi della classi che compongono complessivamente il bean nel seguente modo

```
ed.setEnterpriseBeanClassName("MyBean");
ed.setHomeInterfaceClassName("MyBeanHome");
ed.setRemoteInterfaceClassName("MyRemoteBean");
ed.setPrimaryKeyClassName("MyBeanPK");
```

Per quanto riguarda la persistenza attraverso le seguenti righe di codice è possibile indicare quali campi debbano essere memorizzati nel database

```
Class BeanClass = MyBean.Class;
Field []persistentFields = new Field[n];
PersistentFields[0] = BeanClass.getDeclaredField("beanId");
PersistentFields[1] = BeanClass.getDeclaredField("beanName");
...
```

Tramite l'istruzione

```
ed.setReentrant(false);
```

è possibile avvertire il container che l'oggetto in questione contiene metodi non rientranti. Grazie all'utilizzo di un oggetto di tipo `ControlDescriptor` è poi possibile definire gli attributi relativi alla transazionalità e sicurezza.

Ad esempio

```
ControlDescriptor cd = new ControlDescriptor();
cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);
cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);
```

È da notare il metodo `setRunAsMode()`, con il quale si specifica il modo con cui i metodi del bean potranno agire: in questo caso ogni invocazione altro metodo o risorsa accedute verranno validate in base ai permessi del client.

La gestione delle eccezioni

Dato che dover gestire la persistenza in modo manuale significa dover implementare metodi che secondo il modello RMI sono remoti, uno degli aspetti più importanti di cui si deve tener conto è la gestione delle eccezioni. La prerogativa principale di un metodo remoto è quella di poter generare eccezioni remote, principalmente di tipo `RemoteException`.

Nella versione 1.0 la specifica EJB impone che tutti i metodi remoti possano generare una eccezione di questo tipo; nel caso in cui all'interno dei vari metodi `ejbCreate()` o `ejbStore()` possano essere generate altri tipi di eccezioni, queste dovranno essere catturate e prorogate all'interno di eccezioni remote.

Questa tecnica, tipica nel mondo RMI prende il nome exception wrapping, ed è particolarmente utile in tutti quei casi in cui la business logic del bean si appoggia a sistemi sottostanti per la gestione di particolari risorse, come nel caso di JDBC, JNDI o JavaMail.

Un esempio molto semplice di utilizzo della exception wrapping potrebbe essere

```
public void ejbLoad() throws RemoteException{
   ...
```

```
   try{
      ... Esegue una ipotetica operazione con una sorgente JDBC
   }
   catch(SQLException sqle){
      throw new RemoteException(sqle);
   }
}
```

Con il passaggio alla versione 1.1 di EJB, le eccezioni generate devono essere wrappate in eccezioni di tipo `EJBException: quest'ultima,` discendendo dalla `RuntimeException`, non richiede la dichiarazione della clausola `throws` nella firma del metodo. Il pezzo di codice appena visto nella versione 1.1 potrebbe quindi essere riscritto nel seguente modo

```
public void ejbLoad(){
   ...
   try{
      ... Esegue una ipotetica operazione con una sorgente JDBC
   }
   catch(SQLException sqle){
      throw new EJBException(sqle);
   }
}
```

Il wrapping dovrà essere attuato in modo da propagare una `EJBException` (o `RemoteException`) quando si sia verificato un problema in uno degli strati software sottostanti (JDBC, JNDI, JMS o altro), mentre si dovrà generare una eccezione proprietaria nel caso di un errore o problema nella business logic.

Questa regola vale per le checked exceptions, dove si può controllare l'eccezione tramite il costrutto `try-catch`. Per le eccezioni unchecked il metodo remoto propagherà sempre una eccezione remota di tipo RemoteException.

La persistenza del Bean e l'interfacciamento con il database

Per poter gestire i dati relativi al bean memorizzati nel database, si deve utilizzare una connessione verso il database, connessione che può essere ottenuta direttamente tramite le tecniche standard JDBC, oppure ricavata per mezzo di JNDI da una sorgente dati opportunamente predisposta.

Utilizzando questa seconda soluzione, si potrebbe definire nel bean un metodo `getConnection()` in modo da utilizzarlo negli altri metodi del bean per poter leggere e scrivere i dati nel database, potrebbe essere così definito

```
private Connection getConnection() throws SQLExcepion {
```

```
   try{
      Context context= new InitialContext();
      DataSource source= (DataSource)context.lookup(SourceName);
      return source.getConnection();
   }
   catch(NamingException ne){
      throw new EJBException(ne);
   }
}
```

dove `SourceName` è il nome JNDI dato alla sorgente dati: ogni bean infatti può accedere ad una serie di risorse definite all'interno del JNDI Environment Naming Context (ENC), il quale viene definito al momento del deploy del componente tramite il cosiddetto deployment descriptor.

Questa organizzazione, introdotta con la versione 1.1 della specifica, è valida non solo per le sorgenti JDBC, ma anche per il sistema JavaMail o JMS: di fatto rende il componente ulteriormente indipendente dal contesto di esecuzione, rimandando al momento del deploy la sua configurazione tramite semplici file XML editabili con il tool di deploy utilizzato.

Ad esempio, supponendo che sia

```
SourceName = "java:comp/env/jdbc/myDb";
```

allora nel file deployment descriptor XML si avrebbe

```
...
<resource-ref>
   <description>DataSource per il MyBean</description>
   <res-ref-name>jdbc/myDb</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
...
```

La modalità da seguire nel caso di EJB 1.0 è differente: pur essendo più standard per quanto riguarda la parte JDBC, vincola il codice Java prodotto.

Nel metodo `getConnection()` in questo caso si utilizzerà l'`EJBContext` per poter accedere al nome della sorgente JDBC:

```
private Connection getConnection() throws SQLExcepion {
   Properties Props = context.getEnvironment();
   String JdbcUrl = Props.getProperty("jdbcUrl");
   return DriverManager.getConnection(JdbcUrl);
}
```

Dato che il deploy in EJB 1.0 deve avvenire tramite un programma Java apposito, comunemente detto `MakeDD.java`, si dovrà provvedere ad aggiungere in tale classe qualcosa del tipo

```
Properties Props = new Properties();
Props.put("jdbcUrl","jdbc:<subprotocol>:<name>");
```

Ciclo di vita di un Entity Bean

Alla luce della approfondita analisi dedicata agli entity bean, sia CMP che BMP, può essere utile affrontare più in dettaglio il ciclo di vita di questi componenti: questo dovrebbe consentire di avere chiaro quale sia il significato dei vari metodi visti fino a questo punto, per comprendere come e quando i vari metodi siano invocati.

Stato di pre-caricamento

In questo stato il bean ancora non esiste come entità astratta, ma piuttosto come collezione di file .class e di descriptor files: devono essere forniti al server la primary key, la remote e la home interface, oltre a tutti i file generati i modo automatico dal deploy.

Pooled State

Nel momento della partenza del server, questo carica in memoria tutti i bean di cui sia stato effettuato correttamente il deploy, posizionandoli nel pool dei bean pronti. In tale fase viene creata un'istanza del componente utilizzando il metodo `bean.newIstance()`, il quale a sua volta invoca il costruttore del bean: come accade in RMI, anche in questo caso per il bean non deve essere specificato nessun costruttore, e l'invocazione avviene su quello di default.

In questa fase tutte le variabili d'istanza assumono il loro valore di default, ed il container assegna l'`EntityContext` tramite il metodo `setEntityContext()`.

In questo momento il componente è disponibile per essere utilizzato quando ci sarà bisogno di servire le richieste del client. Tale situazione perdurerà fino a quando non verrà invocato un metodo di ricerca.

Nessun bean in questo caso è assegnato ad un `EJBObject`, e non contiene informazioni significative.

Ready State

Il passaggio al Ready State può avvenire o per esplicita chiamata di un metodo di ricerca da parte del client o per creazione diretta per mezzo dell'invocazione del metodo create.

Nel secondo caso per prima cosa viene creato un `EJBObject`, lo skeleton dell'oggetto secondo la terminologia RMI, ed assegnazione di un bean prelevato dal pool. In questo caso l'invocazione del metodo create da parte del client, viene propagata all'`ejbCreate()`: al suo termine una chiave primaria viene generata ed assegnata all'EJBObject. In questo momento sono effettuate tutte le operazioni di preparazione dei dati sul database in uno dei due modi, a seconda della politica scelta per la gestione della persistenza (CMP/BMP). Successivamente il controllo passa al metodo `ejbPostCreate()` per terminare la fase di inizializzazione.

Infine il client riceve lo stub dell'oggetto remoto, con il quale potrà effettuare tutte le invocazioni da remoto in perfetto accordo con quanto avviene nel modello RMI.

Leggermente differente è invece il caso relativo al passaggio al ready state tramite attivazione. In questo caso il bean viene prelevato dal pooled state dove era finito in seguito ad una passivazione: benché il bean non fosse presente in memoria, il container aveva mantenuto un legame fra lo stub del client e l'EJBObject: durante l'attivazione quindi i dati sono prelevati dal database e rassegnati al bean appena creato prima che questo sia accoppiato con l'EJBObject.

Analogamente il passaggio inverso, ovvero dal ready al pooled può avvenire al verificarsi di vari eventi: sia su invocazione diretta del client sia in conseguenza di una passivazione attuata dal server. Anche in questo caso i metodi `ejbStore()` ed `ejbPassivate()` sono invocati in modo da mantenere sincronizzato il bean con i record del database. La sequenza precisa con cui tali invocazioni sono effettuate dipende dalla implementazione del server.

I Session Beans

Dovendo dare una definizione concisa ed esauriente di session bean si potrebbe dire che si tratta di componenti remoti manipolabili dal client secondo le ormai note interfacce home e remote come per gli degli entity, rispetto ai quale differiscono per una caratteristica fondamentale: mentre gli entity possono essere considerati come una rappresentazione di una struttura dati, i session incorporano al loro interno delle funzionalità o servizi. Per questo a volte si dice che inglobano la business logic del client spostandola sul server remoto. In definitiva quindi il client potrà utilizzare entity bean per accedere a dati remoti, ed invocare i servizi di uno o più session bean per manipolare tali dati.

Esistono due tipi di session bean, gli stateless e gli stateful e si tratta di due componenti molto diversi per scopo e funzionamento: i primi sono componenti senza stato, mentre i secondi consentono di memorizzare uno stato fra due invocazioni successive dei metodi remoti da parte del client.

Stateless Beans

Uno stateless bean non offre nessun mantenimento dello stato delle variabili remote fra

due invocazioni successive dei metodi del bean da parte del client. L'interazione che si crea quindi fra il client e il bean è paragonabile a ciò che accade fra un web browser e un componente CGI server side: anche in quel caso infatti non vi è nessuna memorizzazione dello stato, se non tramite cookie o altre tecniche analoghe.

Tali bean quindi non dispongono di un meccanismo atto a tenere traccia dei dati relativi ad un determinato client: questo è vero esclusivamente per le variabili remote, e non a quelle di istanza interne alla classe. Ad esempio è possibile tramite una variabile di istanza realizzare un contatore del numero di invocazioni di un determinato metodo: ad ogni invocazione, ogni metodo potrà incrementare tale contatore anche se non è possibile ricavare l'identità del client invocante.

Tale separazione fra client e bean remoto impone quindi che tutti i parametri necessari al metodo per svolgere il suo compito debbano essere passati dal client stesso al momento dell'invocazione.

Un bean di questo tipo non ha quindi nessun legame con il client che lo ha invocato: durante il processo di swapping dei vari oggetti remoti, diverse istanze potranno essere associate ad un client piuttosto che ad un altro. Uno stateless quindi è spesso visto a ragione come un componente che raccoglie alcuni metodi di servizio.

Gli stateless offrono una elevata semplicità operativa ed implementativi, unitamente ad un buon livello di prestazioni.

Invocazione di metodi

Il meccanismo di base con cui sono invocati metodi remoti di un session è molto simile a quanto visto fino ad ora per gli entity. La differenza più lampante e forse più importante è quella legata alla diversa modalità con cui il bean può essere ricavato da remoto. Il metodo `getPrimaryKey()` infatti genera una `RemoteException` dato che un session non possiede una chiave primaria. Con la specifica 1.0 questo dettaglio era lasciato in sospeso, ed alcuni server potevano restituire un valore null.

Il metodo `getHandle()` invece restituisce un oggetto serializzato che rappresenta l'handle al bean, handle che potrà essere serializzato e riutilizzato in ogni momento: la condizione stateless infatti consente di riferirsi genericamente ad un qualsiasi bean del tipo puntato dall'handle. Per accedere all'oggetto remoto associato al bean si può utilizzare il metodo `getObject()` della interfaccia `Handle`

```
public interface javax.ejb.Handle{
   public abstract EJBObject getEJBObject() throws RemoteException;
}
```

Comportamento opposto è quello degli stateful, dove un handle permette di ricavare l'istanza associata al componente per il solo periodo di validità del bean all'interno del container.

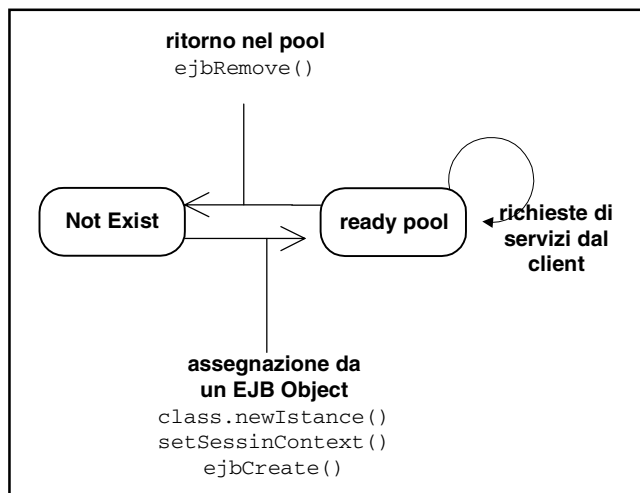
Se il client distrugge esplicitamente il bean, tramite il metodo `remove()`, o se il componente stesso supera il tempo massimo di vita, l'istanza viene distrutta, e l'handle diviene inutilizzabile. Una successiva invocazione al metodo `getObject()` genera una `RemoteException`.

Ciclo di vita di uno Stateless Bean

Dato che uno stateless bean non rappresenta una particolare astrazione dati, né mantiene uno stato fra due invocazioni successive del client, il ciclo di vita per questi componenti risulta essere molto semplificato.

Gli stati che lo compongono sono solamente due, ed il passaggio da uno all'altro coinvolge un numero molto ridotto di operazioni: il primo, *not-exist state*, corrisponde alla non esistenza di alcun componente all'interno del container. Il secondo stato è invece il *ready-pool state*, e corrisponde allo stato in cui il bean è pronto per l'invocazione da parte del client.

Un bean entra nel ready-pool su richiesta del container al momento del bisogno: molti server pre-instanziano un numero arbitrario di componenti al momento della inizializzazione del container, prima che un qualsiasi client ne richieda l'utilizzo; quando il numero dei bean istanziati risulta essere insufficiente, il server ne istanzia altri. Viceversa, quando il numero dei bean allocati è superiore alle reali necessità, il container rimuove dal ready-pool uno o più bean senza effettuare una particolare distinzione su quali eliminare, visto che tutti i componenti sono equivalenti fra loro.

Figura 17.11 – *Ciclo di vita di un session bean stateless*

Il passaggio verso il ready-pool avviene secondo una procedura ben precisa, che vede tre distinte fasi: per prima cosa il bean viene istanziato per mezzo del metodo `Class.newIstance()` corrispondente ad una specie di istanziazione statica del componente.

Successivamente tramite il metodo `SessionBean.setSessionContext()` al bean viene associato un contesto (classe `EJBContext`), che verrà mantenuto per tutto il suo periodo di vita.

Dato che uno stateless bean non prevede nel suo ciclo di vita lo stato di passivazione su memoria secondaria, il context può essere memorizzato indifferentemente in una variabile persistente o non persistente; la direttiva Sun invita i vari costruttori alla prima soluzione.

Alla terminazione del metodo `ejbCreate()` si considera terminato il processo di creazione del componente. Tale metodo viene invocato una sola volta durante il ciclo di vita del componente in corrispondenza del passaggio nello stato di ready-pool.

Quindi nel momento in cui il client invoca il metodo `create()` della home interface non si ha nessuna ripercussione sul componente che esiste già nello stato di ready-pool: non si ha l'invocazione del metodo `ejbCreate()` e si passa direttamente alla creazione di una istanza sul client.

Quando il bean si trova nello stato ready-pool è pronto per servire le richieste dei vari client: quando uno di questi invoca un metodo remoto dell'`EJBObject` relativo, il container associa alla interfaccia remota un bean qualsiasi prelevato dal pool per tutto il periodo necessario al metodo di svolgere il suo compito.

Al termine della esecuzione il bean viene disassociato dal bean, in netta contrapposizione con quanto avviene sia con gli entity bean che con gli stateful, dove il bean resta associato allo stesso EJBObject per tutto il tempo in cui il client ha bisogno di interagire con il bean: questo fatto si ripercuote positivamente sulle prestazioni complessive come sulla quantità di memoria occupata dal sistema.

Sempre a causa della mancanza negli stateless di un meccanismo di persistenza dei dati, non verranno mai invocati i metodi in callback `ejbActivate()` ed `ejbPassivate()`.

In definitiva il processo di istanziazione di un bean di questo tipo è molto più semplice rispetto agli altri casi: quello che però non cambia è la modalità con cui il client ricava un reference remoto. Ad esempio

```
Object obj = jndiConnection.lookup(jndi_bean_name);
MyBeanHome mbh =(MyBeanHome)
PortableRemoteObject.narrow(obj, MyBeanHome.class);
MyBean mb = mbh.create();
```

Il passaggio inverso, da ready-pool a not-exist, avviene quando il server non necessità più della istanza del bean ovvero quando il numero di bean nel pool è sovradimensionato rispetto alle necessità del sistema.

Il processo comincia dall'invocazione del metodo `ejbRemove()`, al fine di consentire tutte le operazioni necessarie per terminare in modo corretto (come ad esempio chiudere una connessione verso una sorgente dati).

L'invocazione da parte del client del metodo `remove()` invece non implica nessun cambiamento di stato da parte del bean, ma semplicemente rimuove il reference dall'`EJBObject`, che tra le altre cose comunica al server che il client non necessita più del reference.

È quindi il container che effettua l'invocazione dell'`ejbRemove()`, ma solamente al termine del suo ciclo di vita. Durante l'esecuzione di tale metodo il contesto è ancora disponibile al bean, e viene rilasciato solo al termine di `ejbRemove()`.

Stateful Beans

Questi componenti sono una variante del tipo precedente e per certi versi possono essere considerati una soluzione intermedia fra gli stateless bean e gli entity.

La definizione che descrive i session come oggetti lato server funzionanti come contenitori della business logic del client, è sicuramente più adatta al caso degli stateful che non degli stateless (per i quali è forse più corretto il concetto di services component).

Questo significa che un determinato bean servirà per tutta la sua vita lo stesso client, anche se il server manterrà sempre attivo un meccanismo di swap fra le varie istanze virtuali dello stesso bean.

Essendo uno stateful dedicato a servire uno e sempre lo stesso client, non insorgono problemi di accesso concorrente.

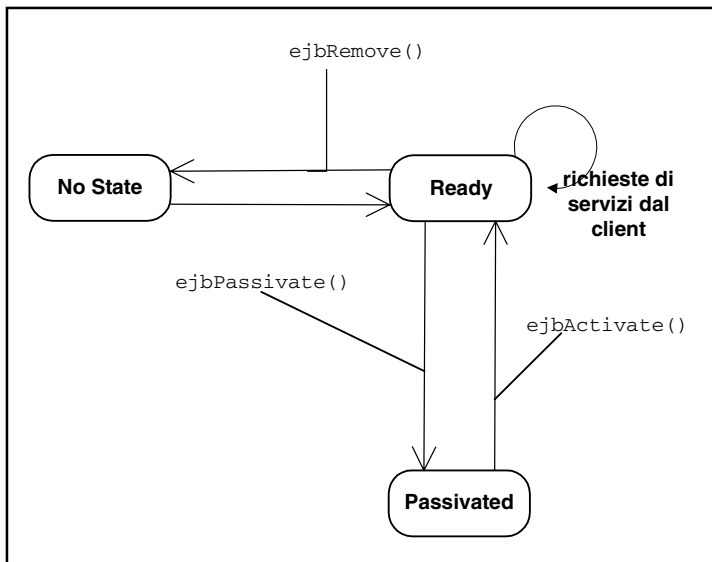
Quindi la differenza che sussiste fra un stateless ed un stateful è che il primo è molto vicino ad essere una raccolta di metodi di servizio, mentre il secondo invece rappresenta l'agente sul server del client.

Ciclo di vita di uno Stateful Bean

Nel ciclo di vita di uno stateful non è presente nessun meccanismo di pool del bean: quando un componente viene creato ed assegnato al client, continua a servire lo stesso client per tutto il suo periodo di vita.

Le diverse implementazioni dei server in commercio utilizzano però tecniche di ottimizzazione effettuando swapping in vari modi, anche se in modo del tutto trasparente, dato che in tal senso nella specifica ufficiale non è presente nessuna indicazione.

Il legame diretto e permanente che si instaura fra il client e l'`EJBObject`, in realtà viene virtualizzato quando durante i lunghi periodi di inattività il bean viene dereferenziato ed eliminato dal garbage collector. Per questo motivo è fornito un meccanismo di persistenza (salvataggio o passivazione e caricamento da memoria o riattivazione) al fine di mantenere il cosiddetto conversational state fra il client ed il bean.

Figura 17.12 – *Gli stati del ciclo di vita di uno stateful sono tre: not exist, ready e passivated*

Il bean ha la percezione del passaggio da uno stato ad un altro solo se implementa l'interfaccia `SessionSynchronization`, che fornisce un set di metodi di callback per la notifica degli eventi al bean, cosa particolarmente utile nel momento in cui si debbano gestire le transazioni.

Gli stati del ciclo di vita di uno stateful sono tre: *not-exist*, *ready* e *passivated*. Il primo è praticamente identico al caso degli stateless e corrisponde allo stato precedente alla istanziazione quando il bean è rappresentato da un insieme di file sul file system.

Quando il client invoca il metodo `create()` della home interface, il container comincia la fase di creazione del componente che inizia il suo ciclo di vita.

Per prima cosa il server crea un'istanza del bean tramite il metodo `newIstance()`, ed assegna un contesto tramite il `setSessionContext()`. Infine il container invoca il metodo `ejbCreate()` al termine del quale il componente, pronto per essere utilizzato, passa nello stato di ready.

Durante questa fase il componente è libero di rispondere alle invocazioni da parte del client, di accedere ad esempio a risorse memorizzate su database, o di interagire con altri bean.

Il passaggio nello stato di passivated può avvenire dopo un periodo più o meno lungo di inutilizzo da parte del client. In questo caso viene rimosso dalla memoria principale (ready state) e reso persistente tramite un qualche meccanismo dipendente dal server (ad esempio tramite serializzazione).

Tutto ciò che non dovrà essere reso persistente, le variabili transient o non serializzabili, dovrà essere messo a null prima della memorizzazione del bean, al fine di impedirne una errata invocazione da parte del client.

Infine se il componente va in time out durante lo stato di passivated, semplicemente verrà eliminato così come verranno eliminati tutti i riferimenti nella memoria.

Il passaggio di stato inverso, da passivated a ready, avviene quando un client effettua una invocazione di un metodo remoto di un bean reso persistente.

In questo caso il componente viene deserializzato e reso disponibile: anche in questo caso si tenga presente che normalmente tutte le variabili primitive numeriche sono inizializzate a zero, mentre le altre non serializzabili a null.

In questo caso la specifica lascia del tutto in sospeso questo aspetto (lasciato quindi al costruttore del server), per cui è bene utilizzare il metodo `ejbActivate()` per effettuare una corretta inizializzazione delle variabili della classe.

Un esempio completo

Vista la vastità dell'argomento si limiterà l'attenzione alla sola presentazione del codice, tralasciando le spiegazioni relative alle fasi di installazione e deploy all'interno dell'application server. Chi lo desiderasse potrà approfondire tali aspetti facendo riferimento ai numerosi documenti reperibili sui siti web delle case produttrici dei vari application server.

Il caso in esame è quello di un carrello virtuale della spesa che permetta l'acquisto di oggetti da parte di un client che si colleghi al server remoto. Come si è avuto modo di fare in precedenza per semplificare la comprensione dei vari aspetti, verranno fatti molti paralleli al caso dell'acquisto online tramite browser e protocollo HTTP.

L'esempio è composto da quattro bean, due session e due entity, in modo da prendere in esame tutte le possibili casistiche. Si analizzeranno brevemente anche alcuni semplici client per evidenziare come un bean possa essere gestito da remoto.

Il codice delle varie classi è organizzato in modo che ogni package contenga un caso particolare (entity CMP, entity BMP, session stateless e session stateful), privilegiando la chiarezza e pulizia, anche se l'organizzazione potrà risultare in qualche caso leggermente ridondante.

Entity Bean

L'entity bean in esame, rappresentato dalla classe `MokaUserBean`, ha il compito di memorizzare alcune informazioni relative all'utente che effettua le operazioni di acquisto di prodotti da remoto. Questa classe è stata definita come base dalla quale creare il BMP ed il CMP. Essa contiene esclusivamente la business logic che sarà presente sia nella versione BMP ed in quella CMP.

Questo piccolo ma utile trucco è reso possibile grazie ad una modifica introdotta con la specifica 1.1: in questo caso il metodo create restituisce sempre una istanza della home interface, indipendentemente dal fatto che si tratti di un CMP o di un BMP. Nel primo caso il valore ritornato sarà ignorato del tutto, mentre nel secondo verrà utilizzato dal codice successivo alla invocazione. Dato che in Java non è possibile l'overload dei metodi con differenti valori di ritorno, la scelta fatta dai progettisti di EJB permette di realizzare strutture pulite con il minimo del lavoro.

Le informazioni relative all'utente sono rappresentate dalle due variabili `userId` e `userCredit`. La prima memorizza un codice univoco con cui l'utente viene identificato, mentre la seconda rappresenta il credito dell'utente: allo scopo di concentrare l'attenzione solo sugli aspetti più significativi di EJB, si è scelto di utilizzare un portafoglio a punti il credito di ogni singolo utente, tralasciando i dettagli relativi alla valuta utilizzata. Tale semplificazione potrebbe essere del tutto plausibile in una community web dove ogni singolo utente compra e vende beni utilizzando punti guadagnati tramite la fornitura o scambio di servizi o beni.

La business logic comune è rappresentata dai due metodi `withdrawCredit()` e `rechargeCredit()` tramite i quali il client può diminuire il credito tramite la riduzione dei punti (conseguente ad una operazione di prelievo) o aumentarlo (ricarica del punteggio). Ecco una loro possibile implementazione

```
public int withdrawCredit(int amount) throws UserAppException{
   if (amount > userCredit){
      throw new UserAppException("Si è tentato di addebitare una
                                 cifra maggiore di quella disponibile");
   }
   userCredit-=amount;
   return userCredit;
}

public int rechargeCredit(int amount){
   userCredit+=amount;
   return userCredit;
}
```

Si noti come in `withdrawCredit()`, prima di procedere al prelievo, venga effettuato un controllo sul credito disponibile ed eventualmente sia generata una eccezione specifica della applicazione. Dato che questi metodi rappresentano la business logic del bean dovranno essere esposti tramite l'interfaccia remota `MokaUserRemote`:

```
public interface MokaUserRemote extends EJBObject {
   public void setUserCredit(int newUserCredit) throws RemoteException;
      public int getUserCredit() throws RemoteException;
      public void setUserId(java.lang.String newUserId) throws RemoteException;
```

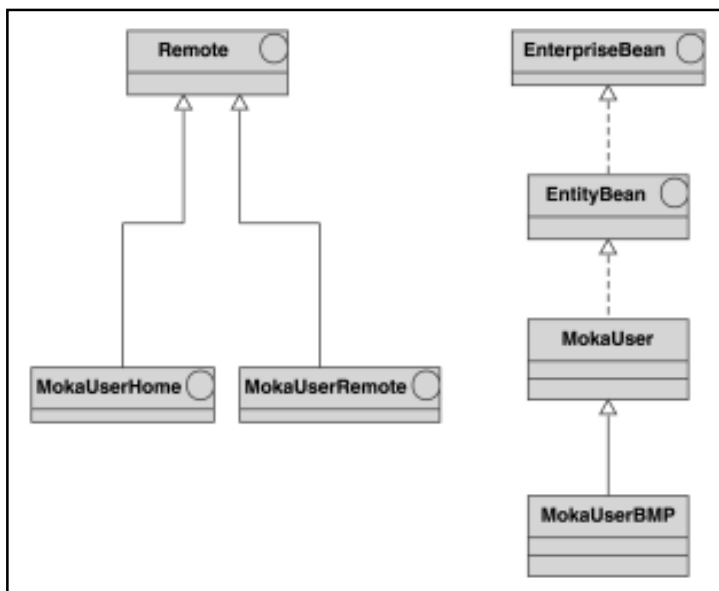
```
    public java.lang.String getUserId() throws RemoteException;

    // business methods
    public int withdrawCredit(int amount) throws UserAppException,
                                                 RemoteException;
    public int withdrawCredit (int amount) throws RemoteException;
}
```

Si noti anche in questo caso la dichiarazione delle eccezioni generabili dai due metodi di business logic. Gli altri metodi della classe sono i metodi (`getXXX` e `setXXX`) per accedere alle due variabili del bean, e a quelli relativi alla gestione del ciclo di vita. Anche in questo caso non vi sono particolari note da evidenziare.

CMP, BMP e ciclo di vita

Dal bean `MokaUser` si estendono le due classi `MokaUserBMP` e `MokaUserCMP` che rappresentano rispettivamente la versione CMP e BMP del bean.

Dato che la business logic resta invariata (metodi `withdrawCredit()` e `withdrawCredit()` non impattano la modalità di gestione della persistenza), tali classi ridefiniranno esclusivamente i metodi relativi alla gestione del ciclo di vita e di ricerca.

Figura 17.13 – *Organizzazione delle classi per l'entity bean BMP*

Ecco ad esempio il metodo `ejbCreate()` nella versione BMP, ovvero nel caso in cui si debba provvedere manualmente all'inserimento tramite SQL dei dati in un database relazionale

```
public String ejbCreate(String uid, int credit) throws CreateException {
   super.ejbCreate(userid, usercredit);
   try {
      //Prima controlla se l'oggetto esiste già
      ejbFindByPrimaryKey(userid);
      //in tal caso genera una eccezione
      throw new DuplicateKeyException("Primary key già presente");
   }
   catch(ObjectNotFoundException e) {}

   // si può procedere alla creazione
   Connection connection = null;
   PreparedStatement statement = null;
   String sqlstm = "";
   try {
      connection = dataSource.getConnection();
   sqlstm = "INSERT INTO user (UserId, UserCredit) VALUES (?,?)";
      statement = connection.prepareStatement(sqlstm);
      statement.setString(1, uid);
      statement.setInt(2, credit);

      if (statement.executeUpdate() != 1)
         throw new CreateException("Errore in aggiunta riga");

      statement.close();
      statement = null;
      connection.close();
      connection = null;
      return userid;
   }
   catch(SQLException e) {
      throw new EJBException("Errore SQL " + sqlstm + " - "+ e.toString());
   }
   finally {
      try {
         if (statement != null) {
            statement.close();
         }
      }
      catch(SQLException e) {}
      try {
         if (connection != null) {
            connection.close();
         }
      }
```

```
      catch(SQLException e) {}
   }
```

Analogamente il metodo `ejbREmove()` effettuerà l'operazione opposta di cancellazione dal database del record corrispondente alla chiave del bean

```
public void ejbRemove(String uid) throws RemoveException {
   super.ejbRemove();
   Connection connection = null;
   PreparedStatement statement = null;
   String sqlstm="";
   try {
      connection = dataSource.getConnection();
      sqlstm="DELETE FROM user WHERE UserId = ?";
      statement = connection.prepareStatement(sqlstm);
      statement.setString(1, userid);

      if (statement.executeUpdate() < 1) {
         throw new RemoveException("Errore nella rimozione del bean");
      }

      statement.close();
      statement = null;
      connection.close();
      connection = null;
   }
   catch(SQLException e) {
      throw new EJBException("Errore SQL " + sqlstm + " - " + e.toString());
   }
   finally {
      try {
         if (statement != null) {
            statement.close();
         }
      }
      catch(SQLException e) {
      }
      try {
         if (connection != null) {
         connection.close();
      }
      catch(SQLException e) {
   }
}
```

Si noti in entrambi i casi il modo con cui sono gestite le eccezioni a seconda del tipo di errore generatosi: per tutti i problemi a livello applicativo si genera una

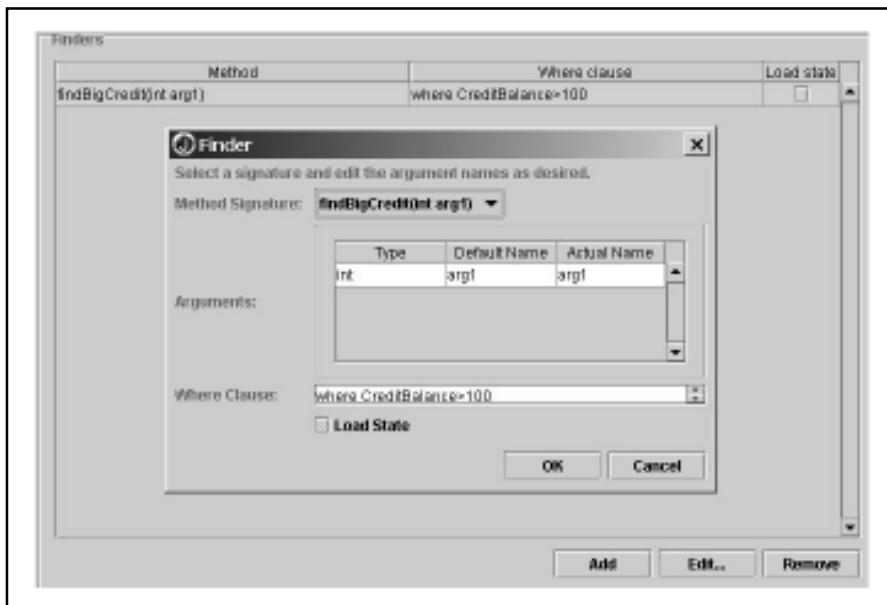
`ApplicationException`, mentre le `SQLException` sono wrappate in una `EJBException`.

Metodi di ricerca

Nel caso di un entity CMP, non deve essere scritta alcuna riga di codice per implementare i vari metodi di ricerca (per chiave o secondo criteri particolari), dato che vengono realizzati al momento del deploy, e per questo non se ne troverà traccia all'interno del bean.

Dovrebbe essere piuttosto intuitivo come il server implementi il metodo di ricerca per chiave primaria (dato che la logica di questa operazione è sempre la stessa), mentre molto meno ovvio è capire come possano essere definite le regole tramite le quali indicare la logica di funzionamento degli altri metodi di ricerca.

Anche in questo caso EJB mette a disposizione una serie di strumenti molto potenti ed efficaci: dato infatti che lo scopo principale di un CMP è astrarsi completamente dal database sottostante si è pensato di definire in modo astratto (es. tramite regole basate su SQL e formalizzate in un file XML) come debba essere effettuata la ricerca.

Figura 17.14 – *Il wizard messo a disposizione da JBuilder 4.0 con il quale è possibile definire le regole relative al metodo di ricerca*

Questa parte è fortemente dipendente dal tool di deploy utilizzato: ad esempio in JBuilder, utilizzando i wizard appositi si otterrà la seguente porzione di codice XML

```
<finder>
   <method-signature>findBigCredit(int arg1)</method-signature>
   <where-clause>where CreditBalance&gt;100</where-clause>
   <load-state>False</load-state>
</finder>
```

che significa che il metodo `findBigCredit()` è uno dei metodi di ricerca che dovrà restituire tutti gli elementi (bean) la cui variabile credit sia maggiore del valore 100.

Analogamente utilizzando WebLogic si otterrà un descriptor file che nella parte relativa alla definizione del metodo di ricerca sarà secondo una struttura in simil-XML

```
finderDescriptors ("findBigCredit (double creditGreaterThan)"
                                                "(> credit $creditGreaterThan)" );
end finderDescriptors
```

Nella prossima versione di EJB (2.0) verrà ufficialmente introdotto un modo simile a quello utilizzato attualmente da JBuilder per la definizione delle regole di ricerca tramite script XML.

Nella versione BMP del bean (classe `MokaUserBMP`) i metodi di ricerca dovranno essere implementare manualmente. Ad esempio il metodo di ricerca per chiave primaria potrebbe essere

```
public String ejbFindByPrimaryKey(String key) throws ObjectNotFoundException {
   Connection connection = null;
   PreparedStatement statement = null;
   String sqlstm="";
   try {
      connection = dataSource.getConnection();
      sqlstm = "SELECT UserId FROM user WHERE UserId = ?";
      statement = connection.prepareStatement(sqlstm);
      statement.setString(1, key);
      ResultSet resultSet = statement.executeQuery();
      if (!resultSet.next()) {
         throw new ObjectNotFoundException("Chiave inesistente");
      }
      statement.close();
      statement = null;
      connection.close();
      connection = null;
      return key;
   }
   catch(SQLException e) {
```

```
      throw new EJBException("Errore SQL " + sqlstm " + e.toString());
   }
   finally {
      try {
         if (statement != null) {
            statement.close();
         }
      }
      catch(SQLException e) {}
      try {
         if (connection != null) {
            connection.close();
         }
      }
      catch(SQLException e) {}
   }
}
```

Session Bean

Rimanendo nell'ambito dell'esempio del carrello virtuale e del client che effettua da remoto degli acquisti addebitati sul conto di un particolare utente, il session bean implementerà quelle funzionalità di gestione del carrello della spesa, con la possibilità di aggiungere e rimuovere dei prodotti.

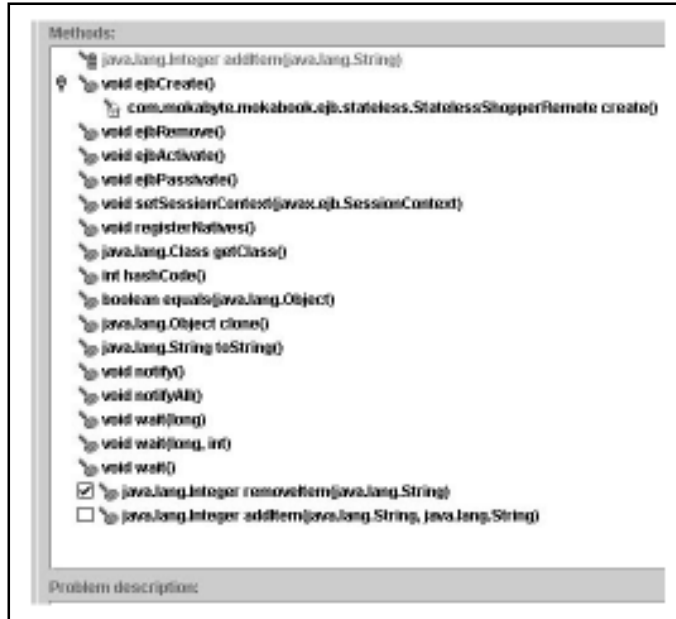
Per fare questo l'interfaccia pubblica del bean offrirà i metodi `addItem()` e `removeItem()` utilizzabili dal client per l'aggiunta e la rimozione di prodotti dal carrello.

Il bean quindi caso svolgerà due funzioni importanti: da un lato fornire il servizio di calcolo del prezzo per ogni oggetto aggiunto al carrello, e dall'altro memorizzare lo stato dei prodotti presenti nel carrello.

Ogni volta il client aggiunge un elemento al carrello della spesa virtuale, verrà effettuata tramite il metodo `getPrice()` una connessione verso un non meglio specificato sistema di service remoto per ottenere il prezzo dell'oggetto in funzione del cliente che ne effettua l'acquisto (ipotizzando un politica di prezzi variabili in funzione dell'utente).

Il contenuto del carrello verrà mantenuto di invocazione in invocazione solo nella versione stateful del bean: nello stateless dovrà essere il client che dovrà tenerne traccia. In questo caso, in analogia con quanto avviene per i sistemi di commercio elettronico via web, è il client che deve tener traccia della sua identità e della persistenza dello stato (ad esempio tramite cookie, o sessioni di vario tipo), il client deve memorizzare il cambiamento della configurazione dopo aver effettuato una invocazione di metodi remoti sul server.

Per questo motivo i metodi `addItem()` e `removeItem()` restituiscono il prezzo dell'oggetto aggiunto al carrello, in modo che il client possa ricavare il prezzo dell'oggetto. Questa organizzazione permette di notare in modo netto ed immediato la differenza fra i due tipi di session bean, dal punto di vista applicativo.

Figura 17.15 – *Solo i metodi che devono essere invocati dal client sono esposti pubblicamente nella interfaccia remota del bean*

Nel caso del bean stateful il container si preoccuperà di mantenere la persistenza delle variabili pubbliche

```
public java.util.Hashtable itemsBasket;
public String clientId;
private int amount;
```

L'implementazione dei metodi per la gestione dei prodotti nel carrello potrebbe essere per `addItem()`

```
public Integer addItem(String productCode){
   Integer price = getPrice(productCode, clientId);
   itemsBasket.put(productCode, price);
   amount += price.intValue();
   return price;
}
```

mentre `removeItem()` potrebbe essere

```
public Integer removeItem(String productCode){
   Integer price = (Integer)itemsBasket.get(productCode);
   itemsBasket.remove(productCode);
   amount -= price.intValue();
   return price;
}
```

Questi due metodi agiscono sulle variabili pubbliche al fine di rendere consistente il contenuto del carrello virtuale, semplicemente per il fatto che sono variabili pubbliche. Il client quindi dopo aver ricavato il riferimento all'interfaccia remoto nel modo consueto

```
Context ctx = new InitialContext();
Object ref = ctx.lookup("StatefulShopper");
statefulShopperHome
= (StatefulShopperHome) PortableRemoteObject.narrow(ref,
                                                     StatefulShopperHome.class);
StatefulShopperRemote Shopper=statefulShopperHome.create("ClientXYZ");
```

potrà procedere a modificare il contenuto del carrello tramite i due metodi appena visti

```
Shopper.addItem("prod_121");
Shopper.addItem("prod_123");
Shopper.addItem("prod_124");
Shopper.addItem("prod_125");
Shopper.removeItem("prod_121");
```

ed eventualmente stampare il conto totale da pagare

```
System.out.println("Importo della merce nel carrello: " + Shopper.getAmount());
```

Nella versione stateless le variabili `itemsBasket`, `clientId` ed `amount` perderanno il loro stato fra due invocazioni successive, e per questo il client ne deve tenere traccia, ad esempio memorizzando ogni volta il totale in una variabile locale

```
Amount+=Shopper.addItem("prod_121", "ClientXYZ");
Amount+=Shopper.addItem("prod_123", "ClientXYZ");
Amount+=Shopper.addItem("prod_124", "ClientXYZ");
Amount+=Shopper.addItem("prod_125", "ClientXYZ");
Amount-=Shopper.removeItem("prod_121");
```

Il client a questo punto potrà ad esempio stampare il conto totale accedendo direttamente alla variabile `Amount`

```
System.out.println("Importo della merce nel carrello: " + Amount);
```

Riconsiderando l'esempio appena visto può apparire più chiaro quando sia indicato utilizzare un bean stateless piuttosto che uno stateful: quest'ultimo, consentendo di gestire il cosiddetto conversational state, è indicato in quei casi in cui le successive invocazioni del client devono essere fatte in funzione di quanto avvenuto in precedenza (si immagini una serie di operazioni di acquisto/vendita in cui si debba agire in base al conto totale totalizzato ad ogni istante). L'esempio appena visto troverebbe piena giustificazione in un stateful.

Se invece lo scopo fosse stato esclusivamente quello di mettere a disposizione del client alcuni metodi di servizio come il `getPrice()`, allora uno stateless bean sarebbe stato sicuramente più adatto.

Dall'esempio analizzato emerge anche come lo sforzo principale di Sun sia rivolto a definire una specifica aperta piuttosto che una API o un prodotto fine a se stesso: questo fatto però pur essendo il principale punto di forza di EJB ne rappresenta anche la maggiore limitazione per tutti coloro che si avvicinano a tale tecnologia per la prima volta. Le molte cose da sapere infatti non si limitano solo alla teoria di EJB, ma anche alle procedure di deploy negli application server e quindi nel comportamento di tali prodotti: la scelta di mantenere la trattazione in questo capitolo a un livello piuttosto teorico e astratto, è stata fatta volutamente, demandando la parte pratica a quanto riportato nei vari tutorial dei prodotti commerciali.

Il client

Il client, rappresentato dalla classe `TestClient`, per prima cosa effettua il lookup dell'oggetto remoto ottenendo un riferimento alla home interface `MokaUserHome`:

```
Context ctx = new InitialContext();
Object ref = ctx.lookup("MokaUser");
UserHome = (MokaUserHome) PortableRemoteObject.narrow(ref, MokaUserHome.class);
```

Successivamente effettua una ricerca nel database per verificare se esista un utente associato ad un determinato id.

```
UserRemote = (MokaUserRemote) UserHome.findByPrimaryKey(UserId);
```

Tale operazione restituisce l'interfaccia remota al bean cercato, oppure null: in tal caso si può procedere alla creazione del nuovo utente

```
if (UserRemote == null) {
   UserRemote = UserHome.create(UserId, initCredit);
}
```

Chi volesse approfondire tali aspetti, in [BEA] potrà trovare alcuni interessanti esempi su come gestire automaticamente i valori dei vari id utente, utilizzando le routine che il database engine mette a disposizione.

Una volta ottenuto il riferimento al bean, in un modo o nell'altro, si potranno effettuare delle modifiche al credito invocando in successione i metodi `rechargeCredit()` o `withdrawCredit()`.

Nella parte finale del client viene implementata gestione una minimale concorrente di un bean da parte di più client. Per prima cosa si creano 10 thread, classe `ThreadClient`, che agiranno sullo stesso bean

```
ThreadClient[] Clients = new ThreadClient[10];
ThreadClient TClient;
for (int i = 0; i < Clients.length; i++) {
   UserId = "" + System.currentTimeMillis();    // unique account id
   Clients[i] = new ThreadClient(UserId, UserRemote, 1500);
   Clients[i].start();
}
```

Il metodo `run()` della `ThreadClient` effettua un semplice prelievo dal credito del bean.

```
public void run() {
   try {
      UserRemote.withdrawCredit(credit2Withdraw);
   }
   catch (Exception e) { ... }
}
```

come si può notare non è necessario preoccuparsi dei dettagli implementativi legati alla concorrenza o aspetti simili.

La gestione delle transazioni

L'accesso concorrente da parte di più client sullo stesso set di bean, e quindi ai dati memorizzati nel database, rende necessario un qualche sistema di controllo dei dati.

Visto che la filosofia di base di EJB è quella di semplificare il lavoro dello sviluppatore di bean, anche in questo caso il lavoro "sporco" verrà effettuato dal container, consentendo una volta di più di lasciare il programmatore a concentrarsi solo sulla logica business del componente.

Introduzione alle transazioni: il modello ACID

L'obiettivo primario di un sistema transazionale è garantire che l'accesso concorrente ai dati non porti a configurazioni incoerenti sui dati stessi o sui risultati di tali operazioni. Per ottenere questo obiettivo in genere si fa riferimento al cosiddetto modello ACID,

ovvero una transazione deve essere atomica (Atomic), consistente (Consistent), isolata (Isolated), e duratura (Durable).

Atomic indica che tutte le operazioni che costituiscono la transazione devono essere eseguite senza interruzioni; se per un qualche motivo una qualsiasi delle operazioni dovesse fallire, allora il motore transazionale dovrà ristabilire la configurazione originaria prima che la prima operazione della transazione sia stata eseguita. Nel caso di transazioni sui database, questo significa che ai dati dovranno essere rassegnati i valori iniziali precedenti all'inizio della transazione. Se invece tutte le operazioni sono state eseguite con successo, le modifiche sui dati nel database potranno essere effettuate realmente.

La consistenza dei dati è invece un obiettivo che si ottiene grazie al lavoro congiunto del sistema transazionale e dello sviluppatore: il sistema fa uso sia di sistemi atti a garantire atomicità ed isolamento, sia di controlli sulle relazioni fra le tabelle del database inseriti nel database engine. Lo sviluppatore invece dovrà progettare le varie operazioni di business logic in modo da garantire la consistenza ovvero l'integrità referenziale, correttezza delle chiavi primarie, e così via.

L'isolamento garantisce che la transazione verrà eseguita dall'inizio alla fine senza l'interferenza di elementi esterni o di altri soggetti.

La durabilità infine deve garantire che le modifiche temporanee ai dati debbano essere effettuate in modo persistente in modo da evitare che un eventuale crash del sistema possa portare alla perdita di tutte le operazioni intermedie.

Lo Scope transazionale

Il concetto di scope transazionale è di fondamentale importanza nel mondo EJB, ed indica l'insieme di quei bean che prendono parte ad una determinata transazione. Il termine scope viene utilizzato proprio per dar risalto al concetto di spazio di esecuzione: infatti ogni volta che un bean facente parte di un determinato scope invoca i metodi di un altro o ne ricava un qualche riferimento, causa l'inclusione di quest'ultimo nello stesso scope a cui appartiene lui stesso: di conseguenza quando il primo bean transazionale prende vita, lo scope verrà propagato a tutti i bean interessati dalla esecuzione.

Come si avrà modo di vedere in seguito, anche se può essere piuttosto semplice seguire la propagazione dello scope monitorando il thread di esecuzione di un bean, la propagazione dello scope deve tener conto della politica definita durante il deploy di ogni singolo bean, dando così vita ad uno scenario piuttosto complesso. La gestione della transazionalità di un bean e quindi la modalità con cui esso potrà prendere parte ad un determinato scope (sia attivamente sia perché invocato da altri bean) può essere gestita in modo automatico dal container in base ai valore dei vari parametri transazionali, oppure manualmente nel caso in cui si faccia esplicitamente uso di un sistema sottostante come Java Transaction Api (JTA).

Nel caso in cui si voglia utilizzare il motore transazionali del container EJB, si può facilmente definire il comportamento del bean tramite gli attributi transazionali (tab. 17.1).

Tabella 17.1 – *Gli attributi transazionali in EJB 1.0 e 1.1*

Attributo transazionale	EJB 1.1 valore testuale	EJB 1.0 costanti predefinite
Not Supported	NotSupported	TX_ NOT_SUPPORTED
Supports	Supports	Tx_SUPPORTS
Required	Required	TX_REQUIRED
Requires new	RequiresNew	TX_REQUIRES_NEW
Mandatory	Mandatory	TX_MANDATORY
Never (1.1)	Never	-
Bean Managed (1.0)	-	TX_BEAN_MANAGES

Il valore di tali attributi è cambiato dalla versione 1.0 alla 1.1: se prima infatti si faceva riferimento a costanti contenute nella classe `ControlDescriptor`, con la specifica 1.1 si è passati a più comode stringhe di testo, cosa che permette di utilizzare file XML per la configurazione manuale del bean.

È possibile definire il comportamento sia per tutto il bean che per ogni singolo metodo: questa possibilità sebbene più complessa ed a rischio di errori, permette un maggior controllo e potenza.

In EJB 1.0 per poter impostare uno dei possibili valori transazionali è necessario invece scrivere del codice Java: ad esempio per prima cosa si deve definire un `ControlDescriptor` impostandone opportunamente l'attributo transazionale

```
ControlDescriptor cd = new ControlDescriptor();
cd.setMethod(null);
cd.setTransactionAttribute(ControlDescriptor. TX_ NOT_SUPPORTED);
ControlDescriptor []cds = {cd};
SessionDes.setControlDescriptors(cds);
```

dove `SessionDes` è un `SessionDescriptor` del bean, da utilizzarsi come visto in precedenza.

In questo caso il valore null in `cd.setMethod(null)` imposta l'attributo transazionale per tutto il bean; volendo invece impostare tale valore per un solo metodo si sarebbe dovuto, tramite la reflection, individuare il metodo esatto e passarlo come parametro.

Ad esempio:

```
Class []parameters = new Class[0];
Method method = MyBean.class.getDeclaredMethod("getName", parameters);
cd.setMethod(method);
```

In EJB 1.1 l'utilizzo di script XML rende tale procedura più semplice. Ad esempio, riconsiderando l'esempio visto in precedenza, si potrebbe scrivere

```
<container-transaction>
   <method>
      <ejb-name>BMPMokaUser</ejb-name>
      <method-name>*</method-name>
   </method>
   <trans-attribute>Required</trans-attribute>
</container-transaction>
```

con il risultato di impostare tutti i metodi del bean `BMPMokaUser` al valore transazionale `Required`.

Normalmente, a meno di particolari esigenze, non è necessario né consigliabile gestire direttamente le transazioni: la capacità di poter specificare come i vari componenti possono prendere parte alle varie transazioni in atto è una delle caratteristiche più importanti del modello EJB, presente fin dalla specifica 1.0.

Significato dei valori transazionali

Si vedranno adesso i vari valori transazionali utilizzando per semplicità il loro nome in italiano, intendendo sia le costanti di EJB 1.0 che le stringhe di testo della versione 1.1.

A volte si usa dire che una determinata transazione client è sospesa: con tale accezione si vuole significare che la transazione del client non è propagata al metodo del bean invocato, ma risulta temporaneamente sospesa fino a che il metodo invocato non termina. Si ricordi che per client si può intendere sia una applicazione stand alone ma anche un altro bean.

Figura 17.16 – *Il funzionamento dell'attributo transazionale not supported. In questo caso lo scope della transazione non verrà propagato al bean o ad altri da lui invocati*

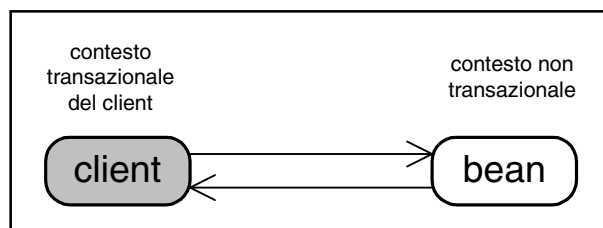
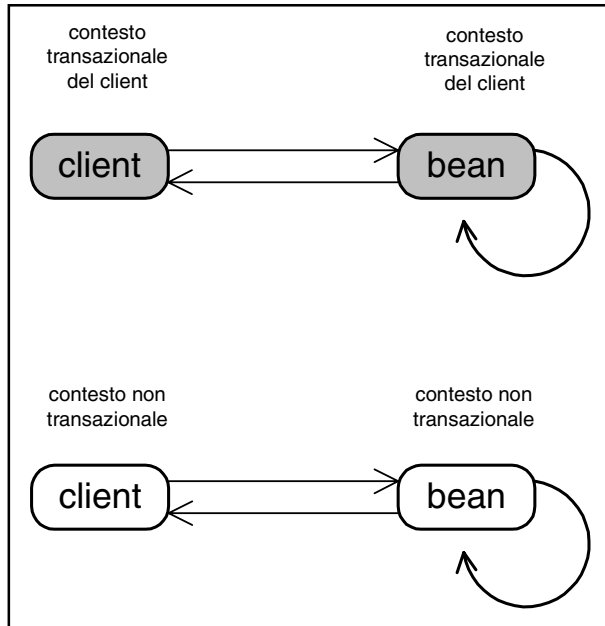



Figura 17.17 – *In questo caso il bean è in grado di entrare nello scope transazionale del client, anche se può essere invocato al di fuori di uno scope*

Not Supported

Invocando all'interno di una transazione un metodo di un bean settato con questo valore, si otterrà una interruzione della transazione; lo scope della transazione non verrà propagato al bean o ad altri da lui invocati. Appena il metodo invocato termina, la transazione riprenderà la sua esecuzione.

Supports

Nel caso in cui il metodo di un bean sia impostato a questo valore, l'invocazione da parte di un client incluso già in uno scope, provocherà la propagazione di tale scope al metodo. Ovviamente non è necessario che il metodo sia necessariamente invocato all'interno di uno scope, per cui potranno invocarlo anche client non facenti parte di nessuna transazione.

Required

In questo caso si ha la necessità della presenza di uno scope per l'invocazione del meto-

Figura 17.18 – *Un bean di questo tipo deve essere eseguito obbligatoriamente all'interno di uno scope: se il client opera in uno scope, il bean entrerà a far parte di quello del client; altrimenti un nuovo scope verrà creato appositamente per il bean*

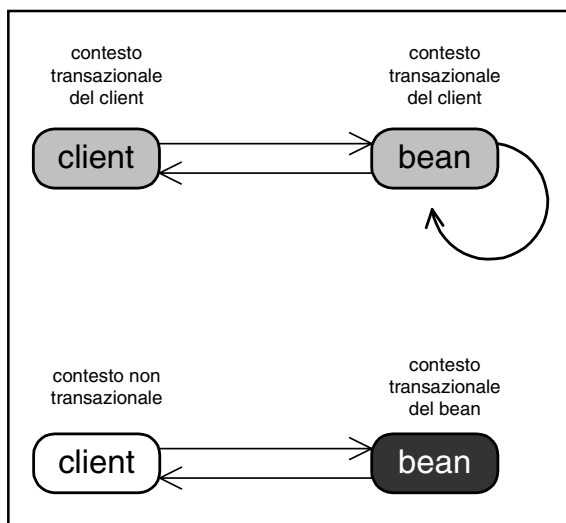
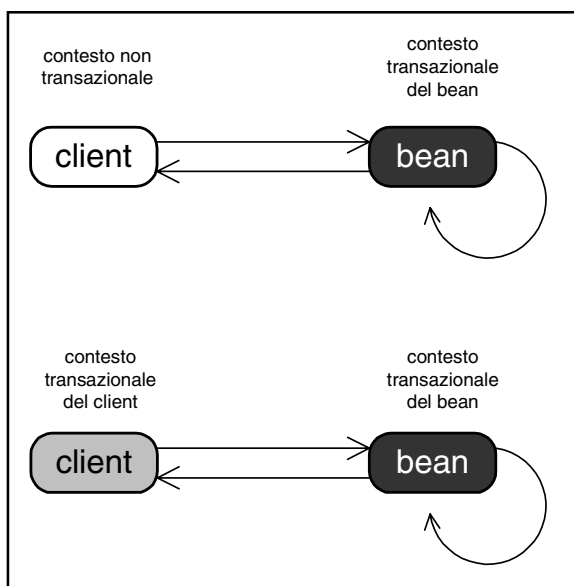



Figura 17.19 – *In questa configurazione il bean creerà sempre un suo nuovo scope*

do. Se il client è coinvolto in una transazione lo scope verrà propagato, altrimenti uno nuovo verrà creato appositamente per il metodo del bean (scope che verrà terminato al termine del metodo).

Requires New

Il bean invocato entra in una nuova transazione, sia che il client faccia parte o meno di una transazione. Se il client è coinvolto in una transazione, quest'ultima verrà interrotta fino al completamento della transazione del bean invocato. Il nuovo scope creato per il bean verrà propagato esclusivamente a tutti i bean invocati dal bean di partenza.

Quando il bean invocato terminerà la sua esecuzione, il controllo ritornerà alla client che riprenderà la sua transazione.

Mandatory

Il bean deve sempre essere parte di una transazione; nel caso in cui il client invocante non appartenga a nessuno scope transazionale, il metodo del bean genererà una eccezione `TransactionRequiredException`.

Figura 17.20 – *È una situazione simile alla required, anche se in questo caso la mancanza di uno scope preesistente nel client provoca la generazione di una eccezione da parte del bean, e non la creazione di uno scope apposito*

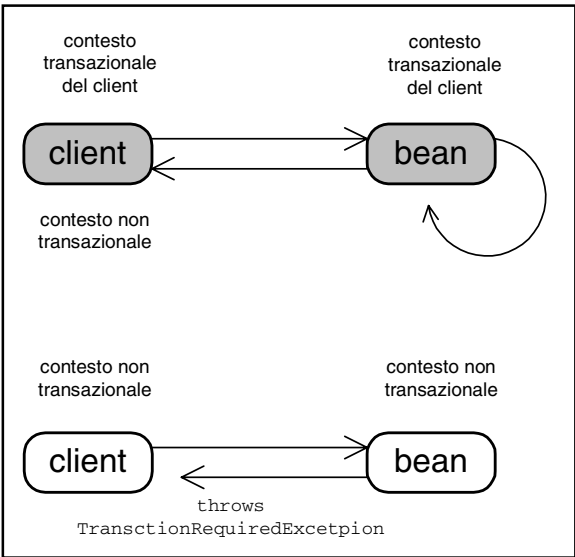



Figura 17.21 – *L'esatto contrario del caso precedente. In questo caso il bean non può assolutamente appartenere a uno scope, pena la generazione di una eccezione*

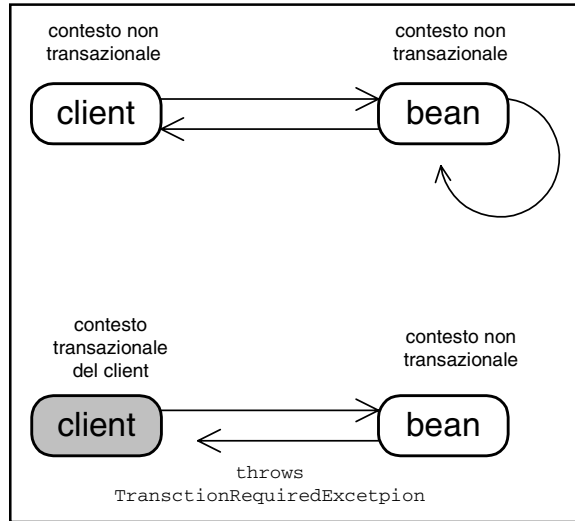
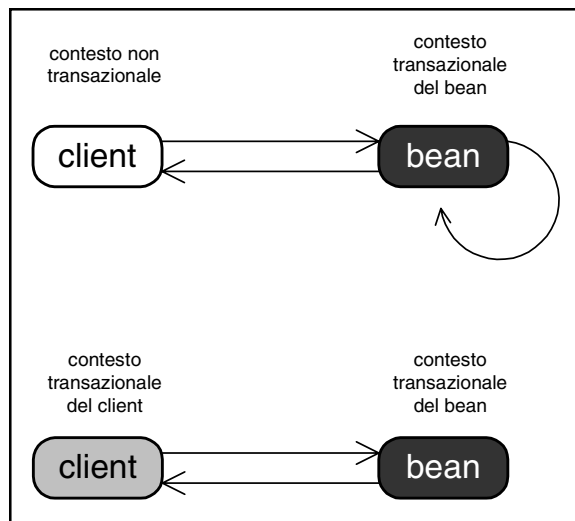



Figura 17.22 – *Il bean opererà sempre in un suo scope personale, gestito direttamente dal bean tramite un transaction engine come JTS*

Never (solo in EJB 1.1)

In questo caso il client invocante non può appartenere a nessun scope transazionale, altrimenti il bean invocato genererà una `RemoteException`.

Bean Managed (solo in EJB 1.0)

In questo caso, possibile solo con la specifica 1.0, si indica che nessun supporto per la transazione deve essere specificato, dato che la gestione verrà effettuata manualmente (tramite ad esempio JTS) all'interno dei metodi del bean.

I vari attributi possono essere impostati in modo autonomo sui vari metodi del bean, ad eccezione dell'ultimo caso, dove l'impostazione a bean managed per anche uno solo dei metodi obbliga a fornire il supporto manuale per tutti i metodi del bean.

Approfondimenti sul database e integrità dei dati

Il concetto forse più importante e critico di un sistema transazionale concorrente è quello dell'isolamento (lettera I del modello ACID). Il livello di isolamento di una transazione in genere è valutabile in funzione del modo in cui riesce a risolvere i seguenti problemi:

- *Dirty reads*: si immagini il caso in cui due transazioni, una in lettura ed una in scrittura, debbano accedere ai medesimi dati. In questa situazione si possono avere incoerenze dei dati, nel caso in cui la transazione in lettura accedesse ai dati appena modificati da quella in scrittura, quando quest'ultima dovesse per un motivo qualsiasi effettuare un rollback riponendo i dati nella configurazione originaria.

- *Repeteable reads*: questa condizione garantisce l'immutabilità dei dati al succedersi di differenti letture all'interno della stessa transazione (ovvero elimina la possibilità che un soggetto possa modificare i dati oggetto della transazione mentre questa è in atto). Una lettura non ripetibile si ha quando una transazione dopo una prima lettura, vedrà alla seguente le modifiche effettuate dalle altre transazioni.

  In genere questa condizione è garantita o tramite un lock sui dati, oppure tramite l'utilizzo di copie dei dati in memoria su cui effettuare le modifiche. La prima soluzione è probabilmente più sicura, anche se impatta pesantemente sulle prestazioni. La seconda invece può complicare molto la situazione, a causa delle difficoltà derivanti dalla necessità di sincronizzare i dati copia con quelli originali.

- *Phantom reads*: letture di questo tipo possono verificarsi quando nuovi dati aggiunti al database sono visibili anche all'interno di transazioni iniziate prima dell'aggiunta dei dati al database, transazioni che quindi si ritrovano nuovi dati la cui presenza è apparentemente immotivata.

Ecco alcune soluzioni comunemente utilizzate per prevenire i problemi di cui sopra:

- *Read locks*: questo blocco impedisce ad altre transazioni di modificare i dati quando una determinata transazione ne effettua una lettura. Questo previene l'insorgere di letture non ripetibili, dato che le altre transazioni possono leggere i dati ma non modificarli o aggiungerne di nuovi. Se il lock venga effettuato su un record, su tutta la tabella oppure su tutto database dipende dalla implementazioni particolare del database.

- *Write locks*: in questo caso, che si presenta tipicamente in operazioni di aggiornamento, le altre transazioni sono impedite di effettuare modifiche ai dati; rappresenta un livello di sicurezza aggiuntivo rispetto al caso precedente, ma non impedisce l'insorgere di letture sporche dei dati (dirty reads) da parte di altre transazioni ed anche di quella in corso.

- *Exclusive locks*: questo è il blocco più restrittivo ed impedisce ad altre transazioni di effettuare letture o scritture sui dati bloccati: le dirty e phantom reads quindi non possono verificarsi.

- *Snapshot*: alcuni sistemi offrono un meccanismo alternativo ai lock, detto comunemente di snapshot (istantanea) dei dati: in questo caso sono create al momento dell'inizio della transazione delle vere e proprie copie dei dati, tali da permettere di lavorare in lettura e scrittura su una copia dei dati. Se questo elimina del tutto il problema dell'accesso concorrente, introduce problemi non banali relativamente alla sincronizzazione dei dati reali con le varie snapshot.

Livelli di isolamento delle transazioni

Per qualificare la bontà di un sistema transazionale in genere si fa riferimento ai cosiddetti livelli di isolamento.

- *Read Uncommitted*: una transazione può leggere tutti i dati uncommitted (ovvero quelli ancora non resi persistenti) di altre transazioni in atto. Corrisponde al livello di garanzia più basso dato che può dar vita a dirty e phantom reads, cosi come possono verificarsi letture non ripetibili.

- *Read Committed*: una transazione non può leggere i dati temporanei (not committed) di altre transazioni in atto. Sono impedite le dirty reads, ma possono verficarsi le letture fantasma e le non ripetibili. I metodi di un bean con questo livello di isolamento non possono leggere dati affetti da una transazione.

- *Repeteable reads*: una transazione non può modificare i dati letti da un'altra transazione. Sono impedite le dirty reads e le letture fantasma ma possono verficarsi le letture non ripetibili.

- *Serializable*: corrisponde al livello massimo di sicurezza, dato che una determinate transazione ha l'esclusivo diritto di accesso in lettura e scrittura sui dati. Si ha la garanzia contro le dirty reads, le letture fantasma e le non ripetibili.

Queste classificazioni corrispondono ai valori definiti come costanti in JDBC all'interno della `java.sql.Connection`.

Gestione del livello di isolamento in EJB

A parte il caso della gestione diretta all'interno del bean delle transazioni, la specifica 1.1 non prevede un meccanismo per l'impostazione tramite attributi del livello di isolamento come invece accadeva per la 1.0.

In questo caso l'isolamento può essere scelto fra quattro livelli ed assegnato ai vari metodi grazie a un oggetto `ControlDescriptor`. Ecco ad esempio una breve porzione di codice che mostra questa semplice operazione

```
ControlDescriptor cd = new ControlDescriptor();
cd.setIsolationLevel(ControlDescriptor.TRANSACTION_SERIALIZABLE);
cd.setMethod(null);
ControlDescriptor []cds = {cd};
SessionDes.setControlDescriptors(cds);
```

In questo caso si è scelto il livello `TRANSACTION_SERIALIZABLE`, mentre nella tab. 17.2 sono riportati tutti i valori disponibili.

In EJB 1.0 è possibile impostare livelli di isolamento differenti per ogni metodo, anche se poi in fase di runtime è necessario che all'interno della medesima transazione siano invocati metodi con lo stesso livello di isolamento.

Tabella 17.2 – *Tramite i valori della* `ControlDescriptor` *è possibile impostare il livello di isolamento voluto*

Livello di isolamento	**Costanti di** `ControlDescriptor`
`Read committed`	`TRANSACTION_READ_COMMITTED`
`Read Uncommitted`	`TRANSACTION_READ_UNCOMMITTED`
`Repeatable reads`	`TRANSACTION_REPEATABLE_READ`
`Serializable`	`TRANSACTION_SERIALIZABLE`

Scelta del livello di isolamento: il giusto compromesso

Al fine di assicurare in ogni istante la correttezza dei dati e delle operazioni svolte su di essi, si potrebbe pensare che la soluzione migliore possa essere l'adozione sempre e comunque di un livello di isolamento molto alto.

Come si è avuto modo già di accennare in precedenza questa non sempre è la soluzione migliore: è vera infatti la regola empirica che vede inversamente proporzionali il livello di isolamento e le performance complessive del sistema. Oltre ad un maggiore numero di operazioni di controllo da effettuare, l'accesso esclusivo in lettura, scrittura o entrambi trasforma gradualmente l'architettura da concorrente a sequenziale.

Purtroppo non esistono regole precise per poter scegliere in modo preciso e semplice quale sia la soluzione migliore da adottare in ogni circostanza e spesso ci si deve basare sull'esperienza valutando in base al contesto in cui si sta operando.

Ad esempio se si è in presenza di un bean che potenzialmente possa essere acceduto contemporaneamente da molti client (è il caso di bean che rappresentano una entità centrale, come la banca nell'esempio visto in precedenza), non sarà proficuo utilizzare un livello alto di isolamento, dato che questo metterà in coda tutte le chiamate di tutti i client.

È altresì vero che, proprio per l'elevato numero di client che accederanno al bean comune, si dovrà fornire un livello di sicurezza relativamente alto, dato che un piccolo errore si potrebbe ripercuotere su un numero elevato di applicazioni client.

La soluzione migliore potrebbe essere quella di separare i vari contesti e di applicare differenti livelli di isolamento per i vari metodi. Ad esempio tutti i metodi che dovranno essere invocati spesso per ottenere informazioni dal bean, ovvero metodi che corrispondono ad operazioni di lettura, potranno essere impostati con un livello basso di isolamento dato che la semplice lettura di un dato non è una operazione pericolosa. In questo caso il valore read uncommitted potrebbe andare bene.

Invece i metodi del tipo `setXXX` possono essere molto pericolosi ed in genere sono essere invocati molto di rado: nell'esempio precedente è probabile che la banca non cambi mai il suo nome. In questo caso deve essere fornito il livello massimo di isolamento visto che le prestazioni non sono un problema dato lo scarso uso dei metodi.

Ovviamente in tutti i casi intermedi si renderà necessaria una scelta adatta al caso specifico.

Gestione esplicita delle transazioni

Sebbene la gestione automatica delle transazione tramite il motore transazionale del server EJB sia la soluzione di gran lunga più semplice e preferibile, non è l'unica possibile: si possono infatti implementare tecniche manuali per ottenere risultati analoghi ma con un maggior controllo sulle singole operazioni effettuate.

A causa della complessità dell'argomento e delle conseguenze che tale scelta comporta la gestione manuale è raramente utilizzata: per questo motivo si affronteranno tali argo-

menti più per scopi didattici che per fornire realmente una conoscenza approfondita di un sistema transazionale.

Si limiterà per quanto possibile la trattazione al caso della gestione delle transazioni in EJB, rimandando alla bibliografia chi volesse approfondire maggiormente gli argomenti relativi alla teoria transazionale.

La specifica EJB prevede che ogni server fornisca un supporto transazionale a livello di metodi del bean. Questa scelta offre già di per se una possibilità di configurazione molto dettagliata, e solo in rari casi si rende necessario scendere ad un livello di dettaglio maggiore.

Inoltre, dato che la definizione del comportamento transazionale del bean avviene in fase di deploy, si può contare su una netta separazione fra il contesto transazionale e quello di business logic. Una eventuale modifica effettuata in fase di deploy infatti permette di modificare radicalmente il comportamento o le prestazioni del bean senza la necessità di dover modificare la business logic del componente.

Bean non transazionali

Non necessariamente un bean deve lavorare in un contesto transazionale: si consideri ad esempio i session bean di tipo stateless, che possono essere visti come collezioni di metodi remoti di servizio; si prenda in esame il caso in cui il bean non effettui nessun tipo di accesso ai dati (ed operi al contempo solamente su variabili locali ai metodi).

Visto che un metodo di uno stateless può essere considerato in definitiva un servizio, mantenerlo fuori dal contesto transazionale (valore `Not Supported`) comporta un indubbio beneficio sulle prestazioni complessive.

La gestione esplicita delle transazioni

La gestione manuale, detta anche esplicita, delle transazioni si basa in genere su un qualche motore sottostante basato sul modello Object Transaction Model (OMT) definito da OMG: nel caso di Java ad esempio si potrebbe utilizzare il sistema JTS, che però per quanto potente offre una API piuttosto complessa e necessita di una chiara visione dell'obiettivo che si vuole raggiungere.

Una soluzione sicuramente più semplice è quella offerta dalla Java Transaction Api (JTA), la quale essendo uscita dopo la versione 1.0 di EJB, non era disponibile inizialmente. Attualmente tale API è da preferirsi sia che si utilizzi la versione 1.0 che la 1.1.

Dato che entrambi i sistemi fanno uso della interfaccia `UserTransaction` i pezzi di codice che si affronteranno saranno validi in entrambi i casi, a parte piccole modifiche di configurazione.

La API JTA è strutturata su due livelli, di cui il più alto e generico è quello normalmente utilizzato dal programmatore di applicazioni.

Per comprendere meglio quando possa essere realmente utile gestire manualmente le transazioni si prenda in esame il caso in cui un client debba eseguire le invocazioni dei due metodi remoti del bean prima di procedere e considerare realmente conclusa l'operazione. La transazione logica in questo caso è rappresentata dall'esecuzione di entrambi i metodi.

Appare quindi piuttosto evidente quale sia il vantaggio di questa soluzione: il client (al solito una applicazione o un altro bean) può effettuare un controllo diretto e preciso sui singoli passi della transazione, avendo sotto controllo l'inizio e la fine della stessa.

Indipendentemente dal tipo di client l'operazione da svolgere è la stessa, e si basa sull'utilizzo di un oggetto `UserTransaction`, anche se il modo per ottenere tale oggetto è piuttosto differente a seconda che il client sia una applicazione stand alone o un bean.

Con il rilascio della piattaforma J2EE Sun ha ufficialmente indicato come una applicazione debba ricavare una ricavare `UserTransaction` grazie a JNDI: quindi nel caso in cui la specifica di riferimento sia EJB 1.1 si potrebbe avere del codice di questo tipo"

```
Context jndiCtx = new InitialContext();
UserTransaction usrTrans;
usrTrans = UserTransaction)jndiCntx.looup("java:comp/UserTRansaction");
usrTrans.begin();
usrTrans.commit();
```

Nel caso di EJB 1.0 invece dato che non è stato esplicitamente indicato il modo per ricavare il riferimento alla `UserTransaction`, si dovrà ricorrere agli strumenti messi a disposizione dal server EJB, come API proprietarie o altri sistemi.

Fortunatamente nella maggior parte dei casi i vari produttori ricorrono ugualmente a JNDI per cui il codice precedente potrebbe diventare

```
UserTransaction usrTrans;
usrTrans
= (UserTransaction) jndiCntx.lookup("javax.trasaction.UserTransaction");
```

Detto questo è interessante notare che anche un bean può gestire le transazioni in modo esplicito. In EJB 1.1 i cosiddetti "bean managed transaction " (in questo caso non è necessario specificare gli attributi transazionali per i metodi del bean) sono quei session bean il cui valore dell'attributo transaction-type sia settato in deploy al valore "bean": ad esempio

```
<ejb-jar>
   <enterprise-bean>
      ...
   <session>
      ...
      <transaction-type>Bean</transaction-type>
      ...
```

Invece gli entity non possono gestire la modalità "bean managed transaction".

In JB 1.0 sia i bean entity che i session possono gestire direttamente le transazioni in modo diretto ed esplicito, a patto che i vari metodi siano impostati con il valore transazionale `TX_BEAN_MANAGED`. Un bean per gestire la propria transazione deve ricavare al solito un reference ad un `UserTransaction` direttamente dall'`EJBContext` come ad esempio

```
public class MyBean extends SessionBean{
   SessionContext ejbContext;
   public void myMethotd(){
      try{
         UserTransaction usrTrans;
         usrTrans = ejbContext.getUserTransaction();
         UsrTrans.begin();
         ...
         UsrTrans.commit();
   }
      catch(IllegalStateException ise){...}
      // gestione di altre eccezioni
      ...
   }
}
```

In questo caso si fa uso del contesto di esecuzione del bean per ricavare direttamente l'oggetto `UserTransaction`: in EJB 1.1 si sarebbe potuto utilizzare indifferentemente la tecnica basata su JNDI come si è visto in precedenza.

Nei session stateless bean (e negli entity in EJB 1.0) una transazione gestita direttamente tramite l'oggetto `UserTransction` deve cominciare e terminare all'interno dello stesso metodo, dato che le stesse istanze dei vari bean in esecuzione sul server possono essere gestite in modo concorrente da più client.

Gli stateful invece, essendo dedicati a servire un solo client per volta, potranno avere transazioni che iniziano in un metodo e terminano in un altro. Nel caso in cui il client, appartenente ad un determinato scope transazionale, invochi su un metodo di un bean in cui sia effettuata una gestione diretta delle transazioni, si otterrà una sospensione della transazione del client fino a quando il metodo remoto del bean non abbia terminato; questo sia che la transazione del metodo del bean inizi all'interno del metodo stesso, sia che sia iniziata precedentemente all'interno di un altro metodo.

Quest'ultima considerazione fa comprendere che la gestione delle transazioni su più metodi sia fortemente da evitare, dato che introduce un fattore di complessità sicuramente molto più grande.

La gestione delle transazioni dal punto di vista del server

Il server EJB per fornire il supporto transazionale ai vari bean deve fornire una imple-

mentazione delle interfacce `UserTransaction` e `Status`: non è necessario quindi che il server supporti il resto della API JTA ne che sia utilizzato il sistema JTS.

L'interfaccia `UserTransaction` ha la seguente definizione

```
interface javax.transaction.UserTransaction {
   public abstract void begin();
   public abstract void commit();
   public abstract void rollback();
   public abstract int getStatus();
   public abstract void setRollbackOnly();
   public abstract void setTransactionTimeout(int secs):
}
```

Ecco il significato dei vari metodi e le funzioni che svolgono:

```
begin()
```

Fa partire la transazione ed associa il thread di esecuzione con la transazione appena creata. Possono essere generate eccezioni di tipo `NotSupportedException` (nel caso in cui il thread sia associato ad un'altra transazione) o di tipo `SystemException` nel caso in cui il transaction manager incontri un problema imprevisto.

```
commit()
```

Completa la transazione associata al thread il quale poi non apparterrà a nessuna transazione. Tale metodo può generare eccezioni di tipo `IllegalStateException` (nel caso in cui il thread non appartenesse a nessuna transazione iniziata in precedenza), oppure `SystemException` se come in precedenza dovessero insorgere problemi inaspettati. Una `TransactionRolledBackxception` verrà generata se la transazione viene interrotta o se il client invoca il metodo `UserTransaction.rollBackOnly()`.

Nel caso peggiore verrà prodotta una `HeuristicRollbackxception` ad indicare l'insorgere di una cosiddetta *decisione euristica*: questo particolare tipo di evento corrisponde alla decisione presa da uno qualsiasi degli elementi che prendono parte alla transazione senza nessuna autorizzazione ne indicazione da parte del transaction manager, effettua una commit o una rollback. In questo caso la transazione perde ogni livello di atomicità e la consistenza dei dati non può essere in alcun modo essere considerata affidabile.

```
rollback()
```

Provoca una rollback della transazione. Una `SecurityException` verrà prodotta se il thread non è autorizzato ad effettuare la rollback; anche in questo caso verrà generata una `IllegalStateException`, se il thread non è associato a nessuna transazione.

`setRollBackOnly()`

Imposta la modalità rollback forzata, provocando obbligatoriamente la generazione di una rollback in ogni caso. Come in precedenza verrà generata una `IllegalStateException`, se il thread non dovesse essere associato a nessuna transazione, una `SystemException` invece verrà lanciata se il transaction manager dovesse incontrare un problema imprevisto.

`setTransactionTimeout(int secs)`

Imposta il tempo massimo entro il quale la transazione debba essere conclusa. Se tale valore non viene impostato, il server utilizzerà quello di default che è dipendente dalla particolare implementazione.

Una `SystemException` verrà lanciata se il transaction manager dovesse incontrare un problema imprevisto

`getStatus()`

Per chi volesse scendere ad un livello più dettagliato di controllo, tramite tale metodo è possibile ricevere un valore intero da confrontare con le costanti memorizzate nella classe `Status` la cui definizione è riportata di seguito

```
interface javax.transaction.Status{
   public final static int STATUS_ACTIVE;
   public final static int STATUS_COMMITTED;
   public final static int STATUS_COMMITTING;
   public final static int STATUS_MARKED_ROLLBACK;
   public final static int STATUS_NO_TRANSACTION;
   public final static int STATUS_PREPARED;
   public final static int STATUS_PREPARING;
   public final static int STATUS_ROLLED_BACK;
   public final static int STATUS_ROLLING_BACK;
   public final static int STATUS_UNKNOWN;
}
```

Il significato di tali valori dovrebbe essere piuttosto intuitivo, per cui non ci dilungheremo oltre nella loro analisi.

Considerazioni sulla gestione manuale delle transazioni

Come si è avuto modo di constatare la gestione diretta delle transazioni sicuramente rappresenta un meccanismo potente per controllare più nel dettaglio i vari aspetti della applicazione sia nella parte di business logic che relativamente alla parte di gestione del ciclo di vita del bean (metodi di callback).

Tale potenza espressiva introduce un livello di complessità che tipicamente non trova giustificazione nella maggior parte dei casi: oltre a richiedere una maggiore conoscenza della teoria transazionale, questa soluzione va contro il principio di separazione fra business logic e motore transazionale come invece avviene quando si rimanda la definizione del comportamento transazionionale alla fase di deploy tramite il deployment descriptor.

Questo è forse il maggior aspetto di cui tener conto, dato che rappresenta anche la maggior potenza del modello EJB.

In definitiva l'esperienza insegna che nella stragrande maggioranza dei casi è meglio affidarsi ai sistemi offerti dal server EJB, eventualmente giocando in modo opportuno sulla configurazione dei vari bean.

La gestione delle eccezioni nell'ambito delle transazioni

Per le eccezioni in Java la classificazione principale vede due principali categorie, le checked e le unchecked. Nell'ambito di EJB ed in particolare della gestione delle transazioni, è utile analizzare le cose sotto un punto di vista leggermente differente, distinguendo fra application exception e system exception (tutte quelle che derivano da `RuntimeException` `RemoteException` e sotto tipi come le `EJBException`).

Nel caso di una system exception viene effettuato un rollback automatico della transazione, cosa che non avviene nel caso di application exception. Questo vale sia per le eccezioni prodotte all'interno dei metodi di business logic che nel caso dei metodi di callback.

In ogni caso la specifica 1.1 impone al server di effettuare un log dell'evento generatosi, anche se poi per la particolare tecnica scelta dipende molto dalla particolare implementazione.

Quando una system exception viene generata il bean viene dereferenziato e rimosso dalla memoria: questa operazione ha importanti conseguenze a seconda che si tratti di un entity di un session stateless o di uno stateful.

Nei primi due casi infatti il client non riceve nessuna informazione circa questo evento, ne il flusso delle operazioni potrebbe risentirne, visto che tali bean non sono dedicati esclusivamente ad un particolare client e, secondo la logica dello scambio di contesto visto in precedenza, servono più client.

Il bean verrà quindi rimosso ed al suo posto ne verrà messo un nuovo appena istanziato.

Nel caso invece di uno stateful le cose sono completamente differenti, visto che esiste un legame stretto fra il bean ed il client: essendo il bean non più presente in memoria, tutte le invocazioni sui metodi da parte del client porteranno a sua volta alla generazione di una eccezione di tipo `NoSuchObjectException`.

Quando una system exception viene generata, ed il bean viene rimosso, il client riceve sempre una eccezione remota (`RemoteExcetpion` o sottotipo a seconda del caso).

Se era stato il client a far partire la transazione, allora l'eccezione generata dal metodo del bean verrà intrappolata e reinnoltrata sotto forma di `Transaction-RolledbackException`. Tutti gli altri casi il container effettuerà la stessa operazione, ma inoltrerà una più generica `RemoteException`.

Le application exceptions invece si verificano in corrispondenza di errori legati alle operazioni di business logic: queste sono in genere direttamente inoltrate al client e non provocano la rollback della transazione; il client può quindi agire in modo da recuperare lo stato effettuato le opportune operazioni di emergenza.

In questo caso il progettista della business logic dovrà fare particolare attenzione a porre i passaggi a rischio (dove potrebbero verificarsi le application exception) prima dell'esecuzione delle operazioni relative al sistema sottostante, al fine di effettuare operazioni inutili o comunque dannose.

Bibliografia

[EJB] *Enterprise Java Bean™ Technology*:
http://java.sun.com/products/ejb/index.html

[EJB2] Richard Monson Haefel, *Enterprise Java Beans*, O'Reilly, 2ª edizione.

[JNDI] *JNDI documentation*: http://java.sun.com/products/jndi/docs.html

[BAS] *Sviluppo di EJB con Jbuilder*: http://community.borland.com/java/all/1,1435,c|3|13,00.html

[BEA] *Documentazione Bea su WebLogic ed EJB*: http://www.weblogic.com/docs/resources.html

[EJBDesign1] Raffaele Spazzoli, *EJB's Design Techniques – I parte*, "MokaByte" 47: http://www.mokabyte.it/2001/01

[EJBDesign2] Raffaele Spazzoli, *EJB's Design Techniques – II parte*, "MokaByte" 48: http://www.mokabyte.it/2001/02

Gestione della sicurezza dal JDK 1.1 a Java 2

di Mauro Molino

Ogni computer connesso alla rete è potenzialmente soggetto a un attacco dall'esterno. Questo è ancora più vero nel caso delle Applet, in cui del software viene scaricato dalla rete. Del resto il codice binario delle Applet viene scaricato automaticamente quando si accede tramite browser a una pagina web che le contiene, quindi non si ha nemmeno il tempo di decidere (a meno che non si imposti nelle opzioni del browser di non scaricare le Applet Java) se scaricarle o meno.

Questi pericoli non si limitano alle Applet, ma si riscontrano anche se si scrivono applicazioni distribuite che fanno uso di *codice mobile* [AdJ] cioè scaricano a runtime dalla rete del codice di cui non vi è traccia sul computer locale.

Essendo Java un linguaggio molto adatto alle applicazioni distribuite, e permettendo il loading dinamico (cioè a runtime) delle classi, esso deve mettere a disposizione un meccanismo per gestire la sicurezza e proteggersi dalle azioni dannose che potrebbero venir compiute da codice remoto.

Il modello Sandbox

Il meccanismo fondamentale è costituito dal modello Sandbox, un ambiente in cui il codice può compiere solo un numero limitato di operazioni che accedono a risorse di sistema come file e connessioni in rete. Secondo questo modello, le applicazioni (codice locale, cioè *trusted*) non sono soggette a restrizioni, mentre le Applet (codice scaricato dalla rete, quindi *untrusted*), essendo eseguite all'interno della Sandbox, possono accedere solo ad un numero limitato di risorse. Questo avrebbe però potuto limitare notevol-

mente l'utilizzo delle Applet per applicativi distribuiti per cui è stato introdotto il concetto di *trusted Applet*, cioè di una Applet con firma digitale, riconosciuta sicura dal sistema e quindi con possibilità di accesso anche a risorse locali.

Questo modello si basa fondamentalmente sull'idea che "prevenire è meglio che curare": di solito una volta che un virus si è diffuso nel sistema è difficile poterlo fermare e comunque il sistema non è più da ritenersi sicuro (anche per un sistema Unix, una volta penetrato da un hacker, sarebbe necessaria una nuova installazione). In questo modo, tramite la Sandbox, fin dall'inizio si proibisce a codice *non fidato* di compiere alcune azioni potenzialmente pericolose.

La sicurezza riguarda diversi aspetti dell'architettura di Java: le varie parti dell'architettura daranno il proprio contributo garantendo che alcuni programmi non riusciranno a compiere azioni dannose. Queste parti sono:

- il linguaggio Java di per sé e la Java Virtual Machine;
- il Class Loader;
- il Class Verifier;
- il Security Manager.

In questo capitolo si vedrà in quali momenti dell'esecuzione di un programma Java, e a che livello, queste quattro componenti operano e interagiscono per garantire la sicurezza. In particolare due di questi componenti sono personalizzabili dal programmatore: il Class Loader e il Security Manager.

Nel caso si utilizzino e si sviluppino solo Applet, non si avrà la necessità di personalizzare tali componenti: anche se lo si volesse fare, un'Applet non potrà utilizzare un proprio Class Loader (diverso cioè da quello del browser, tipicamente chiamato `AppletClassLoader`) né un Security Manager personalizzato (pena, in entrambi i casi, una `SecurityException`), proprio per motivi di sicurezza: in effetti se un'Applet riuscisse a installare un proprio Class Loader o Security Manager, avrebbe la possibilità di caricare in modo personalizzato una certa classe pericolosa o, nel caso del Security Manager, evitare alcuni controlli di sicurezza vitali. Comunque anche in questo caso, capire come funziona la sicurezza in Java, e soprattutto comprendere queste componenti, può senz'altro essere utile per risolvere problemi che si riscontreranno nello sviluppo e nell'esecuzione di programmi Java.

Invece, nel caso si sviluppino applicazioni che scaricano codice dalla rete (codice mobile), non solo si avrà la necessità di scrivere un proprio Class Loader (per poter appunto caricare in memoria il codice scaricato, non presente nel computer locale), ma sarebbe bene (per non dire un obbligo) scrivere anche un Security Manager personalizzato che non permetta al codice ricevuto dalla rete di compiere azioni critiche e potenzialmente

pericolose. Infatti, mentre nel caso delle Applet tutte queste azioni sono automaticamente proibite dalla politica di sicurezza di Java e in particolare dai browser, nel caso delle applicazioni normali non si hanno limitazioni di sorta, ed è quindi teoricamente possibile compiere azioni che danneggino la macchina locale.

La discussione affrontata nel presente capitolo prenderà in considerazione i metodi di sicurezza introdotti quasi subito in Java e presenti quindi in tutte le versioni del linguaggio. Nel corso del capitolo si vedranno anche le importanti innovazioni introdotte in quest'ambito nella piattaforma Java 2.

La sicurezza nel linguaggio e nella JVM

Già a livello di sintassi Java, tramite alcune restrizioni, evita il verificarsi di certi eventi e proibisce la possibilità di compiere alcune azioni. Java è un linguaggio *type safe* — anche se recentemente è stato dimostrato formalmente e con un controesempio che il sistema di tipi di Java non è *sound* cioè corretto, almeno nella versione 1.1 — e il casting a runtime fra i vari tipi è controllato in modo che non venga permessa la conversione fra tipi incompatibili (come ad esempio è possibile, con effetti abbastanza imprevedibili, in C++).

Uno degli errori più frequenti a runtime in linguaggi ad alto livello molto permissivi, come il C ed il C++, è l'utilizzo di puntatori nulli o comunque non inizializzati (cioè contenenti un valore puramente casuale); in questo tipologia di errori rientra anche l'accesso a un elemento di un array oltre i limiti. Questi errori provocano generalmente errori di protezione (nei sistemi operativi più robusti) con conseguente terminazione del programma che li ha causati, o errori irreversibili (nei sistemi operativi più deboli) con conseguente blocco dell'intero sistema.

Java sotto questo aspetto effettua l'*array bounds checking* e controlla che non si utilizzi un riferimento (puntatore) nullo: ogni volta che si accede a un oggetto la JVM controlla che il riferimento che si sta utilizzando per accedervi sia effettivamente valido.

Per quanto riguarda gli errori, a tal proposito, Java utilizza il meccanismo delle eccezioni, che permette di effettuare il *recovery* da una situazione di errore (tra cui, appunto, anche quella dell'utilizzo di un puntatore non valido, o l'accesso oltre i limiti di un array), o comunque, se tale recovery non fosse possibile, di terminare il programma in modo molto più elegante di un semplice *crash* e fornendo alcune informazione e spiegazioni riguardo l'errore che si è verificato. A livello di compilazione, viene forzata la gestione delle varie eccezioni tramite l'utilizzo di un blocco del tipo

```
try {
   azioni che possono generare l'eccezione xxxException
   azioni da svolgere nel caso non si verifichi
} catch (xxxException e) {
   azioni da svolgere per gestire l'eccezione
}
```

Se non si vuole tentare un recovery, un semplice

```
e.printStackTrace()
```

fornirà a schermo lo stack delle chiamate, che senz'altro è molto utile per individuare il punto all'interno del programma che ha causato l'eccezione. Se poi non si volesse nemmeno utilizzare il blocco `try ... catch`, il compilatore obbliga a utilizzare la clausola `throws` nella dichiarazione del metodo che richiama azioni che potrebbero provocare una certa eccezione.

```
public void myMethod(...) throws xxxException {
   azioni che possono provocare un'eccezione xxxException
}
```

In questo modo la gestione di questa eccezione viene delegata ai metodi che richiamano il metodo `myMethod`.

Un altro errore tipico legato all'utilizzo dei puntatori è quello di accedere a una parte di memoria che è già stata rilasciata o di rilasciare più volte la stessa porzione di memoria. Java, sotto questo aspetto, gestisce automaticamente la deallocazione dinamica della memoria (l'allocazione rimane comunque a carico del programmatore) tramite il meccanismo del *Garbage Collector* che deallocherà al momento opportuno la memoria non più utilizzata; un altro meccanismo adottato da Java è quello del *reference counting* tramite il quale si tiene traccia dei vari riferimenti a uno stesso oggetto; in questo modo un oggetto sarà deallocato solo quando non esisterà più nessun riferimento, cioè quando si sarà sicuri che nessuno lo sta più utilizzando.

Fondamentalmente, si è potuto finora constatare che il pericolo principale è l'accesso diretto (o *non strutturato*) alla memoria. Da questo punto di vista, ancora una volta, Java agisce radicalmente, proibendo l'accesso non strutturato alla memoria, non permettendo cioè l'aritmetica sui puntatori. In questo modo non è possibile gestire direttamente indirizzi di memoria e quindi accedere a determinate aree di memoria di cui si potrebbe conoscere l'indirizzo.

A parte il pericolo per il programma in esecuzione, o per l'intero sistema, costituito da un accesso a una regione errata di memoria, l'accesso diretto alla memoria costituisce di fatto un mezzo potente e fondamentale con cui i virus possono accedere a informazioni vitali e a struttura da "infettare". È da notare che l'accesso non strutturato alla memoria è proibito non solo a livello di linguaggio di programmazione, ma anche a livello del set di istruzioni del bytecode; altrimenti la restrizione imposta dal linguaggio sarebbe facilmente raggirabile, scrivendo direttamente il bytecode a mano.

Comunque, anche se un cracker riuscisse a violare tale sistema di sicurezza, si troverebbe in difficoltà nel cercare di accedere alle strutture dati della JVM (ad esempio per accedere al Security Manager e sostituirlo), in quanto l'allocazione delle strutture dati come le

aree di memoria dove saranno allocati lo stack dei thread, lo heap del Garbage Collector, ecc. avviene a runtime. Non c'è quindi traccia di indirizzi di memoria all'interno di un file `.class`. Inoltre dove la JVM allochi tali strutture non è parte delle specifiche della JVM (disponibili pubblicamente), ma è lasciata all'implementatore; quindi tale politica di allocazione è diversa per le varie JVM.

Questo livello di sicurezza viene a mancare quando si utilizzano i metodi *nativi*. Trattandosi di metodi scritti in codice dipendente dall'architettura (e quindi direttamente eseguibile), tali metodi non sono soggetti alle restrizioni della Sandbox. Quindi all'interno di questi metodi è possibile accedere direttamente alla memoria. Soprattutto, poiché i metodi nativi non agiscono attraverso le API di Java, nemmeno il Security Manager (di cui si parlerà in seguito) avrà il controllo su tale codice. Anche in questo caso l'unico modo per evitare che codice remoto possa eseguire del codice nativo, è rappresentato dal proibire ad esso il caricamento di librerie esterne (le dynamic linked library) che sono necessarie per richiamare tali metodi. Questo è quanto viene applicato alle Applet. Anche diverse API di Java sono implementate in codice nativo oppure richiamano metodi nativi, ma si tratta, in questo caso, di codice *fidato*.

Il Class Verifier

Ovviamente la sicurezza a livello di sintassi servirebbe a poco se non ci fossero controlli a livello di bytecode: un cracker potrebbe scrivere direttamente in bytecode e compiere azioni illecite. È necessario compiere un controllo del bytecode prima che questo venga eseguito (si ricorda comunque che la JVM esegue un costante monitoraggio delle varie azioni di un programma Java). Il controllo prima dell'esecuzione si rende necessario per evitare che un programma vada in *crash* in modo inaspettato.

Ogni JVM è dotata di un C*lass File Verifier* [Verif], che assicura che la classe caricata (il suo file `.class`) abbia una determinata struttura interna, conforme allo standard di questi tipi di file. Se il Class Verifier rileva un problema o un errore all'interno del file, verrà lanciata un'eccezione. Questo controllo è necessario in quanto non si può sapere se il bytecode provenga da un compilatore Java o direttamente da un cracker che lo ha scritto a mano.

In un certo senso il Class Verifier non si affida al — o meglio non si fida del — compilatore che, per un eventuale errore interno, non ha notato un errore potenzialmente pericoloso per la robustezza del programma. Ad esempio, nel file `.class` potrebbe esserci un metodo il cui bytecode esegue un salto oltre la fine del metodo: un tale errore manderebbe in crash la JVM. Il Class Verifier verifica quindi l'integrità del codice.

Il momento della verifica della struttura della classe è lasciato all'implementazione; comunque la maggior parte delle JVM esegue questa verifica subito dopo aver caricato in memoria la classe (si veda a tal proposito il paragrafo relativo al Class Loader), una volta per tutte, prima di eseguire il relativo codice. Questo è conveniente: analizzare che la

destinazione dei salti sia una locazione corretta, ad esempio, permette di ignorare questo test a tempo di esecuzione. Il lavoro del Class Verifier può essere suddiviso nelle due fasi che saranno descritte di seguito.

I fase: controlli interni (internal checks)

Di questa fase si è già un po' parlato: viene controllato che il bytecode possa essere eseguito dalla JVM in modo sicuro. Oltre al controllo dei salti, vengono effettuati anche i seguenti controlli:

- ogni file `.class` deve iniziare col *magic number* `0xCAFEBABE`, in modo che si possa riconoscere subito che si tratta di un file `.class` valido;

- ogni componente all'interno del file `.class` contiene le informazioni sul proprio tipo e sulla propria lunghezza (numero di byte che occupa all'interno del file). In questa fase si verifica che questi dati siano coerenti con quello che è presente effettivamente nel file;

- si controlla che ogni componente presenti una propria descrizione (una stringa) che sia ben formata secondo una precisa grammatica;

- inoltre si controlla che certi vincoli imposti dalle specifiche del linguaggio siano soddisfatti; ad esempio tutte le classi, tranne ovviamente `Object`, devono avere una superclasse.

Una volta che la struttura del file `.class` è stata dichiarata consistente, si passa all'analisi del bytecode (questa parte è detta *bytecode verifier*). Durante questa fase vengono esaminate le varie istruzioni controllando la validità dei singoli opcode e operandi (non ci si dilungherà oltre a riguardo di questa parte).

II fase: verifica durante l'esecuzione

Questa seconda fase, effettuata durante l'esecuzione del codice, si occupa del controllo dei riferimenti simbolici. Viene verificato che ogni riferimento ad un'altra classe sia corretto. Questo controllo avviene durante il dynamic linking, che è appunto il processo che si occupa della risoluzione dei riferimenti simbolici in riferimenti diretti: quando infatti si fa riferimento a un'altra classe (tramite un riferimento simbolico) viene cercata la classe (ed eventualmente caricata dal Class Loader) a cui ci si riferisce, e poi viene sostituito il riferimento simbolico con un riferimento fisico (puntatore o offset), le volte successive che si incontra tale riferimento simbolico, non ci sarà bisogno di risolverlo nuovamente.

Durante questo processo il Class Verifier verifica la validità del riferimento (cioè che la classe esista e che l'elemento riferito ne faccia parte).

Questa verifica controlla anche la *compatibilità binaria* fra le varie classi che a causa del caricamento dinamico e dell'eventuale suddivisione in pacchetti delle classi, non può essere assicurata a tempo di compilazione. Il compilatore infatti non controlla le dipendenze fra le classi di pacchetti diversi. Questo tipo di incompatibilità è inoltre dovuta al modo in cui avviene il controllo delle dipendenze: se una classe `A` si riferisce a una classe `B` e si compila `A`, allora viene effettuata una verifica di compatibilità; ma se si modifica e si ricompila la classe `B`, `A` potrebbe non essere più compatibile e il controllo non viene nemmeno effettuato, a meno che `B` a sua volta non faccia riferimento ad `A`; del resto si dovrebbero controllare tutte le classi che utilizzano `B`. Per le regole di compatibilità binaria si veda [Spec].

Da quanto detto si può riassumere che il Class Verifier scopre problemi causati da:

- errori del compilatore;
- cracker;
- incompatibilità binaria.

Il Class Loader

L'utilizzo di Java per scrivere applicazioni distribuite è sempre più frequente; l'indipendenza di Java dall'architettura è fondamentale per quel tipo di applicazioni che si scambiano, oltre ai dati, anche del codice eseguibile. Le applicazioni in Java possono scambiarsi codice eseguibile (detto *codice mobile*) in rete utilizzando il meccanismo personalizzabile del Class Loader [Verif] [Load]. Il Class Loader è utilizzato da Java per caricare in memoria le informazioni (la struttura) di una certa classe.

Java permette di costruire applicazioni estendibili dinamicamente, nel senso che un'applicazione è in grado di caricare, a tempo di esecuzione, nuovo codice e di eseguirlo, incluso del codice che nemmeno esisteva quando l'applicazione è stata scritta. Java infatti effettua il caricamento dinamico delle classi, cioè carica le informazioni relative alle classi, durante l'esecuzione del programma. A tal proposito, vale la pena spiegare quando avviene il caricamento di una classe in memoria. L'approccio di caricare una classe in memoria solo quando serve ottimizza l'utilizzo della memoria, in quanto solo le classi effettivamente utilizzate da un'applicazione saranno caricate. È bene precisare che la dichiarazione di una variabile senza istanziamento

```
myClass myVar;
```

non provoca il caricamento della classe `myClass` in memoria: questo avverrà solo quando un oggetto di tale classe sarà creato, cioè quando la variabile sarà istanziata

```
myVar = new myClass();
```

oppure quando si fa riferimento a un metodo o a un membro statico di tale classe (se ce ne sono). Ad esempio il metodo *static* `forName` di classe `Class` prende come parametro una stringa che specifica un nome di classe e carica tale classe nel programma in esecuzione, restituendo un oggetto `Class` relativo alla classe appena caricata. Se tale metodo non riesce a trovare la classe, lancia un'eccezione `ClassNotFoundException`. Dell'oggetto `Class` così ottenuto si può richiamare il metodo `newInstance` che creerà una nuova istanza di tale classe.

Ad esempio si potrebbe creare una nuova classe solo se è verificata una certa condizione; se tale condizione non sarà verificata, tale classe non verrà mai caricata in memoria

```
if(cond) {
   try {
      Class c = Class.forName("myPack.myClass");
      Object o = c.newInstance();
      ...
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   }
}
```

Del caricamento dinamico delle classi si occupa un oggetto di classe `ClassLoader`, una classe astratta del package `java.lang`, messa a disposizione da Java. Un Class Loader si occupa quindi di importare i dati binari che definiscono le classi (e le interfacce) di un programma in esecuzione.

Nella Java Virtual Machine (JVM) è presente il cosiddetto Class Loader *primordiale* (il Class Loader di default) che viene utilizzato per caricare le classi dal file system locale (comprese quelle delle API di Java). Tale Class Loader fa parte della JVM ed è quindi implementato in C. Il comportamento di questo Class Loader di default sarà quello di cercare un file `.class`, relativo alla classe da caricare, nel file system locale, nei path indicati nella variabile d'ambiente `CLASSPATH`.

Tramite il meccanismo del Class Loader un'applicazione Java viene estesa dinamicamente, cioè a runtime; durante l'esecuzione un'applicazione può determinare di quali ulteriori classi abbia bisogno e caricarle attraverso il Class Loader.

Un'applicazione Java può installare un Class Loader differente da quello primordiale, per caricare le classi in modo personalizzato [Verif], ad esempio scaricandole dalla rete: basterà derivare dalla classe `ClassLoader` e ridefinire certi metodi.

Il metodo della classe `ClassLoader` utilizzato per caricare in memoria una classe è `loadClass`, a cui si deve passare il nome di tale classe come parametro. Tale metodo è

astratto, quindi ne deve essere fornita un'implementazione nelle classi derivate (cioè nei Class Loader personalizzati). All'interno di tale metodo sarà richiamato il metodo `defineClass`, passandogli i dati binari della classe (un array di byte). Tali dati binari corrispondono al contenuto del file `.class` di tale classe. Al termine `loadClass` restituirà un oggetto di classe `Class`.

Esistono due versioni (in overloading) di questo metodo:

```
public Class loadClass(String className) throws ClassNotFoundException {...}

protected abstract Class loadClass(String name,
                                   boolean resolve) throws ClassNotFoundException;
```

Il booleano `resolve` è un flag per indicare se le classi referenziate dalla classe che si sta caricando devono essere risolte (cioè se devono essere caricate a loro volta). Il valore di default è `true`, e viene settato a `false` solo durante le chiamate ricorsive interne.

È importante che un Class Loader si mantenga una tabella (una tabella hash in cui la chiave è il nome della classe e il valore l'oggetto `Class` relativo) con le classi già caricate in quanto uno stesso Class Loader deve sempre restituire lo stesso oggetto `Class` per le classi con lo stesso nome, altrimenti la JVM penserà che ci siano due classi diverse con lo stesso nome, e lancerà un'eccezione. Questo è fondamentale anche perché `loadClass` sarà richiamato ricorsivamente durante la risoluzione di una classe.

All'interno di `loadClass` si dovrebbe sempre tentare di caricare la classe con il Class Loader primordiale e, solo se questo tentativo fallisce, caricare la classe in modo personalizzato.

Un'importante proprietà del meccanismo dei Class Loader è che le varie classi riescono a "vedere" solo le classi caricate col loro stesso Class Loader, e possono accedere solo a queste. Java permette quindi di creare *name space* multipli: per ogni Class Loader la JVM mantiene un name space diverso, contenente i nomi delle classi caricate con quel Class Loader. Quindi una classe che appartiene ad un certo name space può accedere solo alle classi presenti nello stesso name space (cioè quelle caricate con lo stesso Class Loader); questa è una forma di protezione: impedisce a certe classi di interferire con altre.

Un'ulteriore proprietà è che quando una classe `A` (già caricata in memoria) si riferisce a un'altra classe `B`, non ancora caricata in memoria, la JVM cercherà di caricare la classe `B` con lo stesso Class Loader con cui era stata caricata la classe `A`.

Quindi un Class Loader personalizzato ottiene la possibilità di caricare una certa classe prima del Class Loader primordiale.

Ogni volta che un'applicazione carica una classe attraverso un Class Loader personalizzato, si espone a dei rischi [Load]: Java fornisce alle classi appartenenti allo stesso pacchetto particolari privilegi di accesso, non concessi alle classi esterne al package; caricare quindi una classe che si dichiara appartenente al package `java` è rischiosissimo in quanto tale classe, in base ai suddetti privilegi, potrà violare diverse restrizioni di sicu-

rezza. Infatti le classi appartenenti al package `java`, essendo le classi definite da Java, possono praticamente accedere a tutte le risorse del sistema.

I passi che dovrebbero essere eseguiti all'interno del metodo `loadClass` sono i seguenti:

- controllare se la classe richiesta è già stata caricata, e in tal caso restituire l'oggetto memorizzato nella tabella delle classi caricate;
- cercare di caricare la classe dal file system locale, tramite il Class Loader primordiale;
- cercare di caricare la classe dal proprio *repository* (una tabella, scaricando i dati dalla rete, ecc...);
- richiamare `defineClass` con i dati binari ottenuti;
- eventualmente risolvere la classe tramite il metodo `resolveClass`;
- restituire l'oggetto `Class` al chiamante.

La chiamata del metodo `defineClass` provoca anche la verifica del bytecode della classe (in cerca di possibili sequenze di azioni non permesse).

Una volta ottenuto l'oggetto `Class` da `loadClass` è possibile utilizzare il metodo `newInstance` di classe `Class` per creare un'istanza (cioè un oggetto) di tale classe.

Tipicamente le azioni che saranno svolte sono:

```
ClassLoader cl = new MyClassLoader(...);
Class c = cl.loadClass("MyClass");
Object o = c.newInstance();
```

`newInstance` restituisce un `Object`, quindi, per poter utilizzare tale istanza come oggetto di classe `MyClass`, si dovrà effettuare un cast esplicito:

```
((MyClass)o).myMethod();
```

Purtroppo questo provocherà una `ClassCastException`. Questo avviene perché

- solo il Class Loader `cl` è a conoscenza della nuova classe appena creata (caricata);
- due classi sono considerate *castable* solo se hanno una classe in comune. Per quanto già detto le classi caricate con Class Loader diversi appartengono a name space diversi.

Per risolvere il problema si deve fare in modo che la classe caricata derivi da una classe (o implementi un'interfaccia) caricata dal file system locale (il *trusted repository*).

Ad esempio supponendo di avere definito l'interfaccia

```
public interface MyInterface {
   void startMethod();
}
```

e supponendo che la classe `MyClass` implementi questa interfaccia, la sequenza corretta di operazioni da eseguire per effettuare il caricamento dinamico è la seguente

```
ClassLoader cl = new MyClassLoader(...);
Class c = cl.loadClass("MyClass");
Object o = c.newInstance();
((myInterface)o).startMethod();
```

Il meccanismo di utilizzare un Class Loader personalizzato è impiegato dai browser per gestire le Applet. Quando viene aperta una pagina Web contenente un'Applet Java, tale applicazione viene mandata in esecuzione e, al momento in cui è necessaria una classe (che non sia una classe standard di Java), il Class Loader personalizzato scaricherà tale classe dalla rete (chiedendola all'host da cui è stata scaricata la Applet).

Siccome caricare codice dalla rete può portare problemi di sicurezza, le Applet non possono utilizzare un Class Loader personalizzato.

Un esempio di Class Loader

In questo esempio [CLoad] si vedrà l'implementazione di un semplice Class Loader che permetterà di caricare le classi da una certa directory, specificata nel costruttore del Class Loader. I file potranno avere un'estensione diversa da `.class` (ovviamente il contenuto dovrà comunque essere quello di un file `.class` Java) e anche questa sarà specificata al costruttore. Il path specificato potrà essere relativo e in tal caso verrà inteso relativo al `CLASSPATH`.

L'idea è quella di sfruttare questo Class Loader per caricare classi contenute in file con estensioni diverse da quella standard (per un motivo o per un altro), e/o contenute in una directory non specificata nel `CLASSPATH`. L'esempio è puramente dimostrativo, in quanto non vi è un vero e proprio scopo di utilizzo pratico. E comunque il Class Loader presentato è semplice da modificare e senz'altro dovrebbe aiutare a capire il funzionamento del caricamento delle classi in Java. Ecco il codice del Class Loader, in particolare si vedrà subito il metodo `loadClass`.

```
public synchronized Class loadClass(String className,
                                  boolean resolveIt) throws ClassNotFoundException {
```

```
    Class result;
    byte classData[];

    System.out.println("Caricamento della classe: " + className);

    /* Prima si controlla la cache */
    result = (Class)classes.get(className);

    if (result != null) {
        System.out.println("Utilizzata la copia nella cache.");
        return result;
    }

    /* Prima si prova col primordial class loader */
    try {
        result = super.findSystemClass(className);
        System.out.println("Classe di sistema (CLASSPATH)");
        return result;
    }
    catch (ClassNotFoundException e) {
        System.out.println("Non e' una classe di sistema.");
        if (className.startsWith("java.")) {
            System.out.println("Classe pericolosa: " + className);
            System.out.println("Caricamento abortito");
            throw new ClassNotFoundException();
        }
    }

    /* poi si prova a caricare la classe dal path specificato */
    classData = getClassBytes(className);
    if (classData == null) {
        throw new ClassNotFoundException();
    }

    /* viene eseguito il parsing e costruito l'oggetto class */
    result = defineClass(className, classData, 0, classData.length);
    if (result == null) {
        throw new ClassFormatError();
    }

    if (resolveIt) {
        resolveClass(result);
    }

    /* Si aggiorna la cache */
    classes.put(className, result);
    System.out.println("Classe caricata: " + className);
    return result;

}
```

Per prima cosa si controlla se la classe richiesta è già stata caricata (se è presente nella tabella delle classi caricate da questo Class Loader), ed eventualmente si restituisce tale copia. Dopo si cerca di caricare la classe con il Class Loader primordiale; se il tentativo fallisce, prima di andare avanti si controlla che la classe non si dichiari appartenente al pacchetto `java`, per i problemi di sicurezza sopra esposti. Se non ci sono problemi si provvede a caricare la classe dal file con l'estensione specificata, e nel path specificato, tramite la funzione `getClassBytes` (non illustrata qui, ma presente fra i programmi allegati); i byte caricati vengono passati a `defineClass` che provvede a effettuare il parsing e a creare l'oggetto `Class`. Eventualmente la classe viene risolta, se il valore del booleano è `true`. Come si vedrà in realtà non viene chiamato il metodo `loadClass` sopra illustrato, ma la seguente versione

```
/* Questa è la versione chiamata */
public Class loadClass(String className) throws ClassNotFoundException {
   return (loadClass(className, true));
}
```

che richiama la prima versione di `loadClass` col booleano `resolveIt` a `true`. Nelle chiamate ricorsive tale booleano sarà gestito automaticamente.

Il Security Manager

Si è visto che tramite il linguaggio e il Class Verifier è possibile "scartare" codice strutturalmente non corretto e potenzialmente pericoloso; con il Class Loader inoltre è possibile creare name space distinti, evitando così che gli oggetti appartenenti a name space diversi possano interferire l'uno con l'altro. Tuttavia il sistema è ancora aperto ad azioni legali ma potenzialmente dannose, come l'accesso al file system o alle system call. Questo non riguarda le Applet — che possono compiere solo un numero limitato di azioni, e fra queste non ci sono quelle appena citate — ma le applicazioni sulle quali non è attiva nessuna restrizione.

Il Security Manager [AgM] [Pro] permette di definire, e quindi di personalizzare, i limiti (nel senso di confini) della sand box. In questo modo è possibile definire le azioni che possono essere effettuate da certe classi o thread e impedire che ne vengano compiute altre.

Per creare un Security Manager personalizzato si deve derivare dalla classe `SecurityManager`. Si tratta di una classe appartenente al package `java.lang`, astratta fino al JDK 1.1 ma che in Java 2 non lo è più ed è per questo direttamente istanziabile. Grazie a ciò è possibile definire `java.lang.SecurityManager` come Security Manager di default al lancio della JVM nel seguente modo

```
java -Djava.security.manager nomeClasse
```

o istanziarlo direttamente a livello applicativo

```
System.setSecurityManager(new SecurityManager());
```

Un Security Manager è quindi programmato direttamente in Java. In Java 2 è in più possibile modificare il comportamento del Security Manager di default attraverso l'utilizzo delle *policies* (che si vedranno più avanti in questo capitolo). Una volta installato dall'applicazione corrente, tale Security Manager effettuerà un monitoraggio continuo sulle azioni svolte — o meglio che sarebbero svolte — dai vari thread dell'applicazione: le API Java chiedono al Security Manager attivo il permesso di compiere certe azioni, potenzialmente pericolose, prima di eseguirle effettivamente. Per ogni azione di questo tipo esiste nella classe `SecurityManager` un metodo della forma `checkXXX`, dove `XXX` indica l'azione in questione, che viene richiamato prima di compiere tale azione. Ad esempio il metodo `checkRead` viene chiamato dalle API di Java prima di eseguire un'azione di lettura su un file, mentre `checkWrite` prima di eseguire un'azione di scrittura su un file. In Java 2 le chiamate a metodi `checkXXX` si traducono in chiamate a un nuovo metodo `checkPermission`.

L'implementazione di questi metodi stabilisce la politica di sicurezza che verrà applicata all'applicazione corrente; quindi i metodi `checkXXX` stabiliscono se il thread corrente può compiere l'azione descritta da `XXX`. Si tenga comunque presente che un Security Manager non riesce ad impedire l'allocazione di memoria e lo spawning di thread; questo vuol dire che non riesce a controllare i cosiddetti attacchi *denial of service* in cui si impedisce l'utilizzo del computer sommergendolo di richieste (esecuzione di processi e allocazione di tutta la memoria). In questi casi si dovrebbero utilizzare tecniche e controlli costruiti ad hoc.

Alla partenza, un'applicazione non ha nessun Security Manager installato ed è quindi aperta a qualsiasi tipo di azione, in quanto non viene applicata nessuna restrizione sulle azioni. Come già detto, questo non è il caso delle Applet: il browser installa automaticamente un Security Manager, che proibisce diverse azioni alle Applet. Una volta installato il Security Manager rimarrà attivo per tutta la durata dell'applicazione e ovviamente, per motivi di sicurezza, non sarà possibile installare, durante il corso dell'applicazione un altro Security Manager (pena una `SecurityException`). Ciò è coerente con il fatto che un'Applet non può installare un proprio Security Manager (altrimenti la sicurezza non sarebbe più garantita, in quanto l'Applet potrebbe concedere a se stessa qualsiasi tipo di azione). Per installare un Security Manager basterà eseguire le seguenti istruzioni

```
try {
   System.setSecurityManager(new mySecurityManager(...));
} catch (SecurityException se) {
   System.err.println("Sec.Man. già installato!");
}
```

Tipicamente un metodo `checkXXX` ritorna semplicemente se l'azione viene permessa, mentre lancia una `SecurityException` in caso contrario; ad esempio

Tabella 18.1 – *Metodi della classe* `SecurityManager`

sockets	checkAccept(String *host*, int *port*) checkConnect(String *host*, int *port*) checkConnect(String *host*, int *port*, Object *executionContext*) checkListen(int *port*)
threads	checkAccess(Thread *thread*) checkAccess(ThreadGroup *threadgroup*)
class loader	checkCreateClassLoader()
file system	checkDelete(String *filename*) checkLink(String *library*) checkRead(FileDescriptor *filedescriptor*) checkRead(String *filename*) checkRead(String *filename*, Object *executionContext*) checkWrite(FileDescriptor *filedescriptor*) checkWrite(String *filename*)
system commands	checkExec(String *command*)
Interpreter	checkExit(int *status*)
Package	checkPackageAccess(String *packageName*) checkPackageDefinition(String *packageName*)
Properties	checkPropertiesAccess() checkPropertyAccess(String *key*) checkPropertyAccess(String *key*, String *def*)
Networking	checkSetFactory()
Windows	checkTopLevelWindow(Object *window*)

un'implementazione di un metodo `checkRead` potrebbe essere la seguente

```
public void checkRead(String filename) {
   if (azione non permessa)
      throw new SecurityException("Lettura non permessa");
}
```

Quindi quando viene invocato un metodo che utilizza un'API di Java, viene controllato se è installato un SecurityManager e, in caso positivo, viene richiamato il metodo `check` opportuno; ad esempio un'idea di quello che può compiere un'API, prima di uscire dall'applicazione corrente è

```
...
SecurityManager secMan = System.getSecurityManager();
if (secMan != null) {
   secMan.checkExit(status);
}
esegue le azioni per terminare l'applicazione corrente
```

Nella tab. 18.1 sono illustrati i vari metodi presenti nella classe `SecurityManager` relativi alle varie azioni che si possono compiere su determinate risorse di sistema (si rimanda alla documentazione in linea per una descrizione completa dei vari metodi).

Ad esempio, se si volesse proibire ad una certa classe — e alle sue derivate — la possibilità di accedere in scrittura al file system basterà implementare

```
public void checkWrite(FileDescriptor fd) {
   if (Thread.currentThread() instance of NomeClasse)
      throw new SecurityException();
}
```

Oppure se lo si vuole proibire alle classi caricate da un Class Loader personalizzato (ad esempio che sono state scaricate dalla rete)

```
public void checkWrite(FileDescriptor fd) {
   if (Thread.currentThread().getClass().getClassLoader()
                                              instanceof NetworkClassLoader)
      throw new SecurityException();
}
```

Java 2 SDK policy

L'implementazione della sicurezza in Java 2 ha fatto un enorme passo in avanti con l'introduzione dei Policy Files. In realtà non si tratta di una introduzione veramente nuova, ma nel vecchio modello di Sandbox il loro utilizzo prevedeva la scrittura di un proprio Security Manager e di un Class Loader. In JDK 1.1 è necessario creare un nuovo metodo

`checkXXX` nel Security Manager per creare un nuovo permesso. Nella nuova struttura questo non è più necessario. La possibilità di poter personalizzare facilmente le policy di accesso alle risorse, ha permesso di non dover più distinguere accessi di codice locale da codice non locale. Ora anche il codice locale sottostà a controlli di sicurezza così come le Applet o il codice remoto, permettendo un controllo molto più fine e mirato e quindi garantendo anche un maggior livello di sicurezza.

Permessi (Permission)

Un permesso rappresenta un oggetto che definisce l'accesso a risorse. Il permesso viene definito via codice all'interno del programma. Ad esempio possiamo ipotizzare un permesso di accesso in scrittura ad un certo file

```
java.io.FilePermission permesso
= new java.io.FilePermission("/path/file.txt", "write");
```

in cui il path del file rappresenta il *target* e `write` rappresenta l'azione permessa.

È da notare che questa istruzione di per sé non consente alcun accesso se questo non sia specificato anche nel policy file associato. In questo caso ad esempio nel policy file apposito, supponendo che la classe si trovi in "`/classi/`", si avrà

```
grant codeBase "file:/classi/" {
   permission java.io.FilePermission "/path/file.txt", "write";
};
```

Si è visto un esempio di permesso su file, ma naturalmente esiste tutta una serie di altre tipologie di permessi, tra cui

- `java.securityAllPermission` che comprende tutti gli altri;

- `java.awt.AWTPermission` che comprende "target" quali:
  `accessClipboard` per scrittura e lettura da clipboard;
  `listenToAllAWTEvents` per permettere l'accesso a tutti gli eventi AWT;
  `accessEventQueue` per l'accesso alla coda di eventi;

- `java.net.NetPermission` per il quale si cita il target:
  `setDefaultAuthenticator;`

- `java.net.SocketPermission con`
  `accept;`
  `connect;`
  `listen;`

e così via.

Lo scopo comune è comunque quello di poter impedire operazioni illecite e pericolose. Ad esempio il non concedere il permesso listenToAllAWTEvents dell'AWTPermission evita che sia possibile per del codice "virale" stare in ascolto delle azioni (tastiera e mouse) del malcapitato utente, oppure impedire il target setDefaultAuthenticator eviterà che qualcuno possa sostituire del proprio codice all'autenticatore di rete standard per poter captare i dati di autenticazione dell'utente. E così via. Il semplice esempio visto in precedenza mostra quanto sia facile implementare dei criteri di sicurezza il più consoni possibile alla propria situazione, senza dover scrivere del codice inutile. Ma vediamo ora più in dettaglio i Policy Files.

Policy Files

L'utilizzo dei Policy Files permette di ottenere una grande versatilità nella gestione dell'accesso alle risorse. A livello applicativo, il file di policy è rappresentato da un oggetto Policy (package java.security). In questo modo è possibile avere più file diversi da utilizzare nelle varie occasioni.

Intanto va detto che esistono Policy Files di sistema e Policy Files utente. Essi sono editabili, in quanto file di testo, e si trovano in /lib/security/java.policy relativamente al path restituito dalla proprietà java.home per quanto riguarda quello di sistema e .java.policy relativamente al path restituito dalla proprietà user.home per quello utente, che sui sistemi windows è per default c:\{Winnt o Windows}\Profiles\nomeutente per Windows NT e Windows 95 in modalità multiutente e c:\Windows\ su Windows 95 in modalità utente singolo. La locazione dei Policy Files è segnata nel file java.home/lib/security/java.security.

È però possibile aggiungere un riferimento ad ulteriori Policy Files passando alla JVM al momento della chiamata il parametro -Djava.security.policy=policyfilelocation dove naturalmente policyfilelocation rappresenta l'URL del file in esame.

Per capire meglio come funzionino questi file, si darà un'occhiata ai file presenti di default nelle distribuzioni di Java. In particolare, si prendono in considerazione quelli presenti nella distribuzione Java 2 SDK 1.3 scaricabile dal sito www.javasoft.com.

Si parte dal file java.security che, si trova in {java-home}/jre/lib/security/. Il file è ampiamente commentato e abbastanza comprensibile. In questo file, il termine *provider* si riferisce a qualsiasi package che contenga un'implementazione di regole crittografiche per utilizzi tipo certificati digitali. È possibile registrare diversi provider con la sintassi

```
security.provider.<n> = <className>
```

dove <n> è un numero progressivo se si hanno più provider (1 rappresenta il provider

di default) e `<className>` il nome della classe del package sottoclasse di provider, che deve sottostare ad alcune regole imposte dal package `java.security`.

Lo statement

```
policy.provider = <className>
```

definisce la classe che verrà usata come oggetto policy, mentre

```
policy.url.1 = file:${java.home}/lib/security/java.policy
policy.url.2 = file:${user.home}/.java.policy
```

definiscono il file di policy di sistema e quello utente. Anche in questo caso è possibile definirne a piacimento con numero progressivo.

```
policy.allowSystemProperty = true
```

consente invece di poter aggiungere via riga di comando (come visto sopra) degli ulteriori Policy Files. Commentando questa riga si impedisce di farlo. Un'altra opzione importante contenuta in questo file è rappresentata da

```
package.access = <packagename> , <packagename> etc.
```

dove la lista rappresenta i package per i quali deve essere lanciata una eccezione di sicurezza all'accesso se non esplicitamente concessa la `RuntimePermission`.

L'altro importante file presente di default è `java.policy`, presente nella stessa directory del precedente. In questo caso, cioè nel default, questo unico files definisce sia la policy generale sia quella utente ma, come già accennato, è possibile definire file diversi.

Come è possibile vedere, il file si basa su una serie di istruzioni *grant*, che definiscono i permessi di accesso alle varie risorse. Ciò che viene specificato è, da un lato, la locazione del codice (quindi classi) al quale si vuole concedere un privilegio e, dall'altro, i privilegi stessi che vengono concessi. Nell'esempio si vede che alle estensioni standard vengono concessi tutti i privilegi mentre a tutte le altre classi (in realtà si dovrebbe dire a tutti gli altri domini) vengono concessi:

ilpermesso di fermare se stessi (in quanti thread)

```
permission java.lang.RuntimePermission "stopThread";
```

il permesso di stare in ascolto su porte non dedicate (dalla 1024 in su)

```
permission java.net.SocketPermission "localhost:1024-", "listen";
```

i permessi di lettura sulle variabili di ambiente, con le istruzioni

```
permission java.util.PropertyPermission "<variabile>", "read";
```

Questa panoramica rappresenta solo in minima parte le potenzialità offerte da questo modello di sicurezza, ma la logica è sempre la stessa, per cui è semplice imporre tutti i vincoli richiesti a seconda delle occasioni.

Bibliografia

[JSec] Java Security Home Page: `http://www.javasoft.com/security`

[Spec] Java Language Specification:
`http://www.javasoft.com/docs/books/jls/html`

[CLoad] L. BETTINI, *Il Class Loader*, "MokaByte" marzo 1998

[AgM] L. BETTINI, *Agenti mobili in Java (III parte): la sicurezza*, "MokaByte" luglio/agosto 1998

[AdJ] L. BETTINI, *Applicazioni distribuite in Java*, "MokaByte" novembre 1997

[Verif] B. VENNERS, *Security and the Class Verifier*, "JavaWorld" ottobre 1997:
`http://www.javaworld.com`

[Load] B. VENNERS, *Security and the Class Loader architecture*, "JavaWorld" ottobre 1997: `http://www.javaworld.com`

[Pro] *Providing Your Own Security Manager*, dal Tutorial di Java:
`http://java.sun.com/docs/books/tutorial`

[SMJ] Security Managers and the Java 2 SDK:
`http://www.javasoft.com/j2se/1.3/docs/guide/security/smPortGuide.html`

[JSA] Java Security Architecture: `http://www.javasoft.com/security`

Javadoc

di Matteo Manzini

La redazione di una precisa ed esauriente documentazione dell'API di una applicazione o di un insieme di classi è uno dei requisiti fondamentali che ogni programmatore deve soddisfare, affinché il rapporto tra il prodotto da lui scritto e gli utilizzatori che intendono con esso interagire risulti implementato in modo funzionale, il più aderente possibile alle intenzioni del programmatore stesso.

La necessità del programmatore è quindi quella di illustrare nel dettaglio l'interfaccia che la sua applicazione espone: ogni classe, metodo o campo ha caratteristiche proprie e modalità di utilizzo che la documentazione deve riportare.

L'importanza di questa documentazione cresce notevolmente al crescere del numero di classi implementate ma è comunque un requisito dal quale non si può prescindere nemmeno per applicazione minime nella quantità di codice scritto e di funzionalità esposte.

La stessa Sun si manifesta pienamente convinta del ruolo fondamentale svolto, all'interno del contesto generale di produzione di un'applicazione, dalla documentazione dell'API, e lo dimostra attraverso due segnali forti e precisi:

- l'insieme di pagine HTML che accompagna il Java Development Kit sin dalla sua prima versione è voluminoso quanto dettagliato, indispensabile per orientarsi in una collezione di package sempre più numerosa e articolata;

- Javadoc™, lo strumento utilizzato da Sun per produrre la documentazione ufficiale di cui sopra, viene incluso in JDK e di conseguenza distribuito gratuitamente. In particolare, Javadoc è presente a partire da JDK 1.1, ma non riporta una propria versione, riconoscibile in modo indipendente perché specificata nell'eseguibile o nel package di riferimento: la consuetudine è quella di indicare a questo scopo la

stessa versione del JDK nel quale è contenuto (quindi, ad esempio, Java 2 SDK versione 1.2.2 contiene Javadoc versione 1.2.2).

La comunità degli sviluppatori si trova quindi a disposizione, da questo punto di vista, sia il migliore esempio del "prodotto finito" ottenibile, sia lo strumento attraverso il quale tale risultato viene raggiunto (e le indicazioni per usarlo). L'ovvio punto di arrivo che si intende perseguire attraverso una politica di questo tipo è la spinta ad adottare e promuovere il più possibile l'utilizzo di Javadoc.

Come si comporta e cosa può fare Javadoc? In termini generali, lo strumento messo a disposizione da Sun elabora le dichiarazioni e i commenti doc presenti in uno o più file sorgenti Java, producendo un set corrispondente di pagine HTML che descrivono classi, interfacce, costruttori, metodi e campi con indicatori di visibilità public o protected.

In pratica, Javadoc è in grado di riproporre, in un formato facilmente consultabile, lo "scheletro" delle dichiarazioni interne al codice sorgente, completamente spogliato delle complicazioni, in questo caso superflue, dell'implementazione.

Non solo: al fine di rendere l'implementazione davvero utile per l'utilizzatore che intende interfacciarsi con l'insieme di classi in questione, il programmatore è tenuto ad arricchire tale scheletro con chiarimenti, precisazioni ed esempi d'uso. Possiamo definire con l'espressione "commenti doc" gli elementi che consentono di ottenere tale risultato.

Commenti doc

Come noto, Java supporta sia i commenti multilinea (caratteristici di C), delimitati da `/*` e `*/`, che i commenti monolinea (caratteristici di C++), indicati da `//`. Entrambi i commenti appartenenti a queste due categorie, definibili come commenti ordinari, vengono inseriti all'interno del codice sorgente Java con lo scopo di rendere più comprensibili i dettagli implementativi del codice stesso.

Il fattore che assume rilevanza assoluta nel contesto di questo capitolo è però un terzo tipo di commento, quello che abbiamo già indicato come "commento doc" e che viene utilizzato da Javadoc per integrare la documentazione dell'API di ogni applicazione.

I commenti doc iniziano con `/**` (invece del normale `/*`), terminano con `*/` e di solito sono multilinea. Vengono inseriti nel codice sorgente e sono riconosciuti e processati da Javadoc solo se immediatamente precedenti la dichiarazione di una qualunque entità (classe, interfaccia, metodo, costruttore, campo). Ogni commento doc è formato da due sezioni:

- la descrizione, che inizia dopo il delimitatore iniziale `/**` e prosegue fino alla sezione dei tag;
- la sezione dei tag, la quale inizia con il primo carattere `@` che si trova in testa ad una riga (escludendo asterischi e spazi).

Figura A.1 – *Esempio di pagina HTML costruita da Javadoc: il file sorgente Java utilizzato non presenta commenti doc*

È importante sapere che la prima frase di ogni descrizione deve essere una rappresentazione concisa ma completa dell'entità dichiarata. Essa viene infatti pensata e scritta per

figurare anche da sola, in quanto usata come definizione generale e inserita da Javadoc nella sezione dal titolo `Summary`.

I commenti doc sono scritti in HTML. Al loro interno è vivamente sconsigliato l'utilizzo di tag HTML strutturali, come ad esempio `<H1>`, `<H2>` o `<HR>`, i quali potrebbero interferire negativamente con la struttura della pagina HTML già predisposta da Javadoc.

Di seguito alcuni esempi.

Esempio 1: commento doc con una sola sezione, su una sola linea

```
/** Sezione descrittiva del commento doc */
```

Esempio 2: commento doc con una sola sezione, su più linee

```
/**
 * Sezione descrittiva del commento doc
 */
```

Esempio 3: commento doc con due sezioni, su più linee

```
/**
 * Sezione descrittiva del commento doc
 *
 * @tag Commento per il tag
 */
```

Quest'ultimo è il caso maggiormente ricorrente, oltre ad essere il più completo dal punto di vista informativo. In particolare, nell'esempio 3:

- la prima riga rientra rispetto al margine in modo da essere allineata con il codice che segue il commento doc. Inizia con `/**`;
- le righe successive iniziano con `*` (asterisco) e rientrano di uno spazio ulteriore per allineare tra loro gli asterischi su righe successive;
- è preferibile inserire una riga vuota tra la descrizione e la sezione dei tag, e tra bocchi di tag logicamente correlati;
- l'ultima riga inizia con `*/`.

Qualunque sia la forma scelta tra quelle appena presentate, durante l'elaborazione dei commenti doc ogni asterisco iniziale di riga viene eliminato, così come ogni spazio o

tab che lo precede. È inoltre preferibile non avere righe eccedenti la lunghezza di 80 caratteri. Infine, eventuali paragrafi interni al commento doc risultano più facilmente individuabili se separati tramite il tag HTML `<p>`.

Javadoc duplica i commenti doc di metodi che sono implementati da altri metodi o dei quali viene effettuato l'override. Ad esempio, se il metodo `mioMetodo1()` in una certa classe o interfaccia non ha alcun commento doc, Javadoc utilizza il commento doc di `mioMetodo2()`, che `mioMetodo1()` implementa o di cui effettua l'override. Quanto detto si verifica quindi in tre situazioni:

- quando un metodo in una classe effettua l'override di un metodo in una superclasse;
- quando un metodo in una interfaccia effettua l'override di un metodo in una delle interfacce da cui eredita;
- quando un metodo in una classe implementa un metodo in una interfaccia.

Occorre puntualizzare che, in questo caso, i nomi `mioMetodo1()` e `mioMetodo2()` sono funzionali dal punto di vista della chiarezza dell'esempio, ma ovviamente non corretti dal punto di vista del linguaggio, in quanto il nome del metodo che effettua l'override deve essere esattamente lo stesso del metodo che lo subisce. Lo stesso discorso vale per i metodi che implementano altri metodi e per i corrispondenti implementati.

Come già anticipato, esiste un insieme di tag speciali che, se inseriti all'interno di un commento doc, Javadoc riconosce ed elabora. Ognuno di questi tag comincia con `@` e deve trovarsi all'inizio di riga, fatta eccezione per eventuali spazi o simboli di asterisco che possono precederlo.

Di seguito si trovano alcuni tra i tag più interessanti (con i relativi argomenti), accompagnati da una breve descrizione.

```
@author [nome]
```

Aggiunge `Author:` seguito dal nome specificato.

Ogni commento doc può contenere pù tag `@author`, presentati in ordine cronologico.

```
@version [versione]
```

Aggiunge `Version:` seguito dalla versione specificata.

```
@param [nome del parametro] [descrizione]
```

Aggiunge il parametro specificato e la sua descrizione alla sezione `Parameters:` del metodo corrente. Il commento doc riferito a un certo costruttore o a un certo metodo deve obbligatoriamente presentare un tag `@param` per ognuno dei parametri attesi, nell'ordine in cui l'implementazione del costruttore o del metodo specifica i parametri stessi.

```
@return [descrizione]
```

Aggiunge `Returns:` seguito dalla descrizione specificata. Indica il tipo restituito e la gamma di valori possibili. Può essere inserito solamente all'interno del codice sorgente Java, dove deve essere obbligatoriamente usato per ogni metodo, a meno che questo non sia un costruttore oppure non presenti alcun valore di ritorno (`void`).

```
@throws [nome completo della classe] [descrizione]
```

Aggiunge `Throws:` seguito dal nome della classe specificata (che costituisce l'eccezione) e dalla sua descrizione. Il commento doc riferito a un certo costruttore o a un certo metodo deve presentare un tag `@throws` per ognuna delle eccezioni che compaiono nella sua clausola `throws`, presentate in ordine alfabetico.

```
@see [riferimento]
```

Aggiunge `See Also:` seguito dal riferimento indicato.

```
@since [versione]
```

Utilizzato per specificare da quale momento l'entità di riferimento (classe, interfaccia, metodo, costruttore, campo) è stata inserita nell'API.

Funzionamento

Uno dei punti di forza del meccanismo ideato da Sun è probabilmente costituito dal fatto che, al fine di realizzare le proprie elaborazioni, Javadoc si basa sulla presenza del compilatore Javac. In particolare, Javadoc utilizza Javac per compilare il codice sorgente, con lo scopo di mantenere le dichiarazioni e i commenti doc, scartando tutte le parti di implementazione. Così facendo viene appunto costruito quello scheletro al quale si è già accennato in precedenza e che risulta essere la base delle pagine HTML di output.

Figura A.2 – *Esempio di pagina HTML costruita da Javadoc: il file sorgente Java utilizzato presenta commenti doc al suo interno*

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

miopackage
Class MiaClasse

```
java.lang.Object
  |
  +--miopackage.MiaClasse
```

public class **MiaClasse**
extends java.lang.Object

Classe di esempio.

Field Summary

int	miaVariabile01 Variabile di tipo int

Constructor Summary

MiaClasse(int mioParametro)
Costruttore che inizializza miaVariabile01

Method Summary

int	getMiaVariabile01() Restituzione del valore di miaVariabile01
java.lang.String	toString() Rappresentazione della classe

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

miaVariabile01

public int **miaVariabile01**

Variabile di tipo int

Constructor Detail

MiaClasse

public **MiaClasse**(int mioParametro)

Costruttore che inizializza miaVariabile01
Parameters:
mioParametro - il valore per inizializzare miaVariabile01

Method Detail

getMiaVariabile01

public int **getMiaVariabile01**()

Restituzione del valore di miaVariabile01
Returns:
il valore di miaVariabile01

toString

public java.lang.String **toString**()

Rappresentazione della classe
Overrides:
toString in class java.lang.Object
Returns:
il valore di miaVariabile02

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Ciò significa che, in teoria, sarebbe possibile far lavorare Javadoc già dalle fasi di design, nel momento in cui l'interfaccia inizia a prendere corpo ma è ancora assente l'implementazione nella sua totalità.

Tale accorgimento moltiplica i vantaggi se adottato all'interno di un gruppo di lavoro costituito da più individui, soprattutto per i programmatori eventualmente assegnati alla scrittura della parte client: esiste chiarezza sin dal primo istante in merito all'interfaccia che essi andranno a interrogare.

Il concetto può essere illustrato considerando come esempio un'ipotetica applicazione Java–CORBA, per la quale è norma che chi interroga gli oggetti remoti sia interessato solamente all'interfaccia esposta e non all'implementazione. In questo caso, lo sviluppatore che scrive tali oggetti remoti ha la possibilità di produrre la documentazione completa delle API non appena terminata la fase di redazione del file IDL, consegnando a chi si occupa di scrivere il codice di interrogazione tutto ciò di cui questi necessita per iniziare la propria opera (in un formato più leggibile rispetto al linguaggio IDL).

L'utilizzo di Javac da parte di Javadoc è garanzia di produzione di documentazione il più esatta ed affidabile possibile: l'esempio maggiormente significativo da questo punto di vista è la creazione del costruttore di default da parte di Javac e la sua conseguente presenza nella documentazione HTML, anche nel caso in cui tale costruttore non venga indicato esplicitamente dal programmatore nel codice sorgente.

Dati i meccanismi di funzionamento appena descritti, Javadoc può essere sfruttato solo con riferimento a codice Java e non ad altri linguaggi di programmazione.

Utilizzo

Se consideriamo l'aspetto pratico, Javadoc viene lanciato da linea di comando nel seguente modo:

```
javadoc [opzioni] [package] [file sorgenti] [@mieifile]
```

L'ordine degli argomenti non è rilevante; nei successivi paragrafi viene riportato il loro significato.

`opzioni`

Le opzioni possibili sono numerose; di seguito le più interessanti (con relativi argomenti).

- `-public`, che mostra nella documentazione solamente classi, metodi e campi con modificatore di visibilità `public`;

- `-protected`, che mostra nella documentazione solamente classi, metodi e campi con modificatore di visibilità `protected` e `public` (questa è l'opzione di default);

- `-package`, che mostra nella documentazione solamente classi, metodi e campi con modificatore di visibilità `package`, `protected` e `public`;
- `-private`, che mostra nella documentazione tutte le classi, i metodi e i campi;
- `-doclet [nome della classe]`, che specifica la classe del doclet che si intende utilizzare al posto dello standard doclet;
- `-1.1`, che genera la documentazione con l'aspetto e le funzionalità della documentazione prodotta da Javadoc 1.1;
- `-sourcepath [locazione dei sorgenti]`, che specifica il percorso per raggiungere i file sorgenti Java quando, dopo il comando `javadoc`, è presente l'argomento `[package]`;
- `-classpath [locazione delle classi (classpath)]`, che specifica il percorso per raggiungere le classi Java;
- `-J[opzione java]`, che passa l'argomento indicato direttamente alla VM;
- `-d [nome della directory]`, che specifica, in modo assoluto oppure relativo, la directory di destinazione nella quale Javadoc salva i file HTML generati;
- `-version`, che consente di processare il tag `@version` nella documentazione generata;
- `-author`, che consente di processare il tag `@author` nella documentazione generata;
- `-link [url]`, il quale specifica un URL verso un set di documentazione che, prodotto da Javadoc in un momento precedente l'esecuzione corrente, si vuole rendere raggiungibile dalle pagine che Javadoc si appresta a generare;

`package`

Un elenco di nomi di package, separati da spazi. È necessario indicare nello specifico ognuno dei package che si intende documentare, qualificandolo con il nome completo e senza la possibilità di utilizzare wildcard come `*` (asterisco);

`file sorgenti`

Un elenco di nomi di file sorgenti Java, separati da spazi, ognuno dei quali può iniziare

con un path e contenere wildcard come `*` (asterisco);

`@mieifile`

Un tag che specifica uno o più file i quali contengono nomi di package e nomi di file sorgenti Java, in qualunque ordine, uno per ogni riga. Javadoc tratta il contenuto dei file indicati come se ogni singola riga si trovasse direttamente su linea di comando.

Fonti

Javadoc è in grado di produrre output a partire da quattro diverse possibili fonti. La prima di queste fonti, dominante sulle altre in termini qualitativi e quantitativi, è il codice sorgente Java, già ampiamente trattato nel corso del capitolo. Ad esso si possono aggiungere:

- file di commento ai package, che Javadoc provvede a includere nella pagina di descrizione del package da esso stesso generata. Il file di commento a un package deve avere nome `package.html` (uguale per tutti i package) ed essere posizionato nella directory del package, la medesima che contiene i file sorgenti Java. Il contenuto di `package.html` è un unico, ampio commento doc, scritto in HTML, il quale non presenta i separatori caratteristici `/** e */`, e nemmeno l'asterisco in testa ad ogni riga;

- file di commento generale, che Javadoc provvede ad includere nella pagina di descrizione generale da esso stesso creata. Il file di commento generale ha solitamente nome overview.html ed è di norma posizionato nella più "alta" directory interna al percorso contenente i file sorgenti Java (nessuna di queste due disposizioni è obbligatoria). Il contenuto del file di commento generale è un unico, ampio commento doc, scritto in HTML, con le stesse caratteristiche appena viste per `package.html`;

- file diversi non processati che si ritiene opportuno includere nella documentazione e rendere quindi raggiungibili dalle pagine HTML generate da Javadoc: è il caso, ad esempio, di file con immagini, di file sorgenti Java (`.java`) con esempi, di file Java compilati (`.class`) o anche di file HTML il cui contenuto potrebbe essere troppo esteso per un normale commento doc. I file in questione devono essere inseriti in una directory chiamata `doc-files`, la quale può essere subdirectory di una qualunque directory di package. È prevista l'esistenza di una directory `doc-files` per ogni package, e i nomi dei file devono essere esplicitamente indicati all'interno dei commenti doc di riferimento. Javadoc non esamina il contenuto di `doc-files`, limitandosi a copiarlo nella corretta directory di destinazione.

Doclet

Javadoc ha un comportamento di default ben preciso, assegnato e applicato da Sun tramite quello che viene definito standard doclet. Esso stabilisce che venga generata documentazione in formato HTML, la maggior parte della quale è costituita da file che hanno lo stesso nome (con diversa estensione) delle classi o delle interfacce specificate. A queste pagine HTML se ne aggiungono altre la cui denominazione è slegata dal contenuto del codice sorgente e tra le quali possiamo citare, ad esempio, `package-summary.html` (una per ogni package documentato), `overview-summary.html` (una per l'intero insieme dei package documentati), `deprecated-list.html` (una per tutte le voci deprecate) e così via.

Come è facile immaginare, Sun non si limita a imporre lo standard doclet come strumento di modellazione dell'output di Javadoc, ma consente di personalizzarne il contenuto e il formato tramite l'utilizzo dei cosiddetti doclet.

I doclet sono veri e propri programmi scritti in Java, i quali utilizzano l'API doclet al fine di specificare il contenuto e il formato dell'output di Javadoc. Il programmatore ha, a questo scopo, sia la facoltà di scrivere classi completamente nuove, sia la possibilità di modificare lo standard doclet.

Le classi che costituiscono l'API doclet si trovano nel file `lib/tools.jar` all'interno di JDK.

Altri esempi

Esempio 4: codice sorgente

Viene presentato il codice del file `MiaClasse.java`, processato con Javadoc per ottenere la pagina HTML di fig. A.2.

```
package miopackage;

/**
 * Classe di esempio.
 * @author Manzini Matteo
 * @version 1.0
 */
public class MiaClasse {

 /**
  * Variabile di tipo int
  */
  public int miaVariabile01;

 /**
  * Variabile che contiene la rappresentazione della classe
  */
```

```
  private String miaVariabile02 = "public class MiaClasse";

 /**
  * Costruttore che inizializza miaVariabile01
  * @param mioParametro il valore per inizializzare miaVariabile01
  */
  public MiaClasse(int mioParametro) {
    miaVariabile01 = mioParametro;
  }

 /**
  * Restituzione del valore di miaVariabile01
  * @return il valore di miaVariabile01
  */
  public int getMiaVariabile01() {
    return miaVariabile01;
  }

 /**
  * Rappresentazione della classe
  * @return il valore di miaVariabile02
  */
  public String toString() {
    return miaVariabile02;
  }
}
```

Di seguito i passi da compiere per riprodurre la documentazione di alcune classi del JDK. Per tutti gli esempi della sezione corrente valgono le seguenti supposizioni:

- locazione dei file sorgenti Java: `C:\home\src\java\util\*java`;
- directory di destinazione: `C:\home\html`.

Esempio 5: documentazione di uno o più package

Al fine di documentare un package nel suo insieme è necessario che i file sorgenti in esso contenuti si trovino in una directory chiamata come il package stesso. Nel caso di nome di package composto da molteplici elementi separati da punti, ognuno di questi elementi rappresenta una diversa directory (ad esempio, alla classe `Vector` del package `java.util` deve corrispondere il file `\java\util\Vector.java`).

In questo caso è possibile eseguire Javadoc in uno dei seguenti modi:

- muovendosi nella directory superiore alla più alta tra le directory che rappresentano il package in questione e lanciando Javadoc con le opzioni e gli argomenti

appropriati. Esempio:

```
C:> cd c:\home\src
C:\home\src> javadoc -d C:\home\html java.util
```

- lanciando Javadoc da qualunque directory, con le opzioni e gli argomenti appropriati. Esempio:

```
C:> javadoc -d C:\home\html -sourcepath C:\home\src java.util
```

Sia nel primo che nel secondo caso viene generata documentazione in formato HTML per le classi e le interfacce `public` e `protected` dell'intero package `java.util`. La documentazione viene salvata nella directory di destinazione specificata (`C:\home\html`).

Esempio 6: documentazione di una o più classi

È possibile utilizzare il carattere `*` (asterisco) per indicare gruppi di classi. In questo caso si può eseguire Javadoc in uno dei seguenti modi:

- muovendosi nella directory contenente i file sorgenti Java e lanciando Javadoc con le opzioni e gli argomenti appropriati. Esempio:

```
C:> cd C:\home\src\java\util
C:\home\src\java\util> javadoc -d C:\home\html Date.java Hash*.java
```

- muovendosi nella directory superiore alla più alta tra le directory che rappresentano il o i package in questione e lanciando Javadoc con le opzioni e gli argomenti appropriati. Esempio:

```
C:> cd c:\home\src
C:\home\src> javadoc -d C:\home\html java\util\Date.java java\awt\Button.java
```

- lanciando Javadoc da qualunque directory, con le opzioni e gli argomenti appropriati. Esempio:

```
C:> javadoc -d C:\home\html C:\home\src\java\util\Vector.java
```

Nel primo caso viene generata documentazione in formato HTML per la classe `java.util.Date` e per tutte le classi del package `java.util` il cui nome inizia con `Hash`. Nel secondo caso viene generata documentazione in formato HTML per le classi `java.util.Date` e `java.awt.Button`. Nel terzo caso viene generata documentazione in formato HTML per la classe `java.util.Vector`.

Esempio 7: documentazione di package e classi

In questo caso (che riprende gli esempi precedenti, documentando package e classi allo stesso tempo) è possibile eseguire Javadoc nel seguente modo:

- lanciando Javadoc con le opzioni e gli argomenti appropriati. Esempio:

```
C:> javadoc -d C:\home\html -sourcepath C:\home\src
    java.util C:\home\src\java\awt\Button.java
```

Viene qui generata documentazione in formato HTML per tutte le classi del package `java.util` e per la classe `java.awt.Button`.

Jumpstart (JS)

Alcuni degli argomenti che nel corso di questa parte iniziale di approfondimento non hanno trovato il giusto spazio vengono trattati nel riassunto schematico che segue: esso è stato soprattutto preparato come un riferimento essenziale e facilmente consultabile, adatto a chi già conosce i concetti che stanno alla base di Javadoc, ma non ne ricorda tutte le caratteristiche nel dettaglio.

"Jumpstart (JS)" è pensato con una propria autonomia e identità: è questo il motivo per cui alcune delle sezioni già presentate nell'approfondimento (ad esempio "Utilizzo") vengono qui riproposte ("JS: utilizzo").

La parte iniziale rimane dal canto suo un insieme di informazioni ampiamente sufficienti per avvicinarsi per la prima volta allo strumento Javadoc, considerando senza dubbio la documentazione dell'API come parte integrante e rilevante di ogni applicazione o insieme di classi, e non come un semplice accessorio opzionale.

JS: utilizzo

```
javadoc [opzioni] [package] [file sorgenti] [@mieifile]
```

L'ordine degli argomenti non è rilevante:

- `opzioni`: le opzioni possibili sono numerose. Vedere sezione "JS: opzioni";
- `package`: un elenco di nomi di package, separati da spazi. È necessario indicare nello specifico ognuno dei package che si intende documentare, qualificandolo con il nome completo (l'utilizzo di wildcard come `*` non è consentito);
- `file sorgenti`: un elenco di nomi di file sorgenti Java, separati da spazi, ognuno dei quali può iniziare con un path e può contenere wildcard come `*`;

- `@mieifile`: un tag che specifica uno o più file i quali contengono nomi di package e nomi di file sorgenti Java, in qualunque ordine, uno per ogni riga. Javadoc tratta il contenuto di questi file come se fosse direttamente su linea di comando.

JS: file generati

Il comportamento di default di Javadoc prevede l'utilizzo dello standard doclet. Esso genera documentazione in formato HTML, suddivisibile in due gruppi ben distinti:

- file che hanno lo stesso nome (con diversa estensione) delle classi o delle interfacce considerate;

- file che hanno nomi non legati al contenuto dei sorgenti Java.

Tipi di file prodotti dallo standard doclet:

- una pagina per ogni classe o interfaccia documentata (`MiaClasse.html`);

- una pagina per ogni package documentato (`package-summary.html`);

- una pagina per l'intero insieme dei package documentati (`overview-summary.html`), a patto che questi siano almeno due. È consigliata come pagina di partenza nel caso di visualizzazione senza frame HTML;

- una pagina contenente la gerarchia delle classi per l'intero insieme dei package documentati (`overview-tree.html`);

- una pagina contenente la gerarchia delle classi per ogni package documentato (`package-tree.html`);

- una pagina di "utilizzo" per ogni package (`package-use.html`) e una per ogni classe o interfaccia (`class-use/MiaClasse.html`) documentati: descrivono quali package, classi, metodi, costruttori e campi utilizzano parti del package, della classe o dell'interfaccia in questione. Affinché Javadoc possa generarle è necessario specificare l'opzione `-use` (vedere "Opzioni per lo standard doclet");

- una pagina contenente tutte le voci deprecate (`deprecated-list.html`);

- una pagina contenente le informazioni riferite a serializzazione ed esternalizzazione (`serialized-form.html`);

- un indice completo di nomi di classi, interfacce, costruttori, campi e metodi, elencati in ordine alfabetico (`index-*.html`);
- una pagina di aiuto per la consultazione della documentazione (`help-doc.html`);
- una pagina di supporto alla creazione di frame HTML. È consigliata come pagina di partenza nel caso di visualizzazione con frame HTML;
- una pagina di supporto alla visualizzazione della documentazione con frame HTML (`package-list`). ;
- un insieme di pagine contenenti elenchi di package, classi e interfacce, usate quando l'utente sceglie di visualizzare la documentazione con frame HTML (`*-frame.html`);
- uno style sheet (`stylesheet.css`);
- una directory contenente file non processati da Javadoc.

JS: tag per commenti doc

Esiste un insieme di tag speciali che, se inseriti all'interno di un commento doc, Javadoc riconosce ed elabora. Ognuno di questi tag comincia con `@` e deve trovarsi all'inizio di riga, fatta eccezione per eventuali spazi o simboli di asterisco che possono precederlo.

Di seguito è presentato l'elenco dei tag standard per commenti doc: l'ordine qui scelto è quello consigliato per l'utilizzo.

```
@author [nome]
```

Aggiunge `Author:` seguito dal nome specificato.

Può essere inserito solamente all'interno del codice sorgente Java e deve essere usato per ogni definizione di classe o di interfaccia, ma non per metodi o campi considerati singolarmente. Ogni commento doc può contenere molteplici tag `@author`, presentati in ordine cronologico. Javadoc non mostra le informazioni sull'autore, a meno che non sia specificata l'opzione `-author` su linea di comando.

Introdotto in JDK 1.0.

```
@version [versione]
```

Aggiunge `Version:` seguito dalla versione specificata.

Può essere inserito solamente all'interno del codice sorgente Java e deve essere usato per ogni definizione di classe o di interfaccia, ma non per metodi o campi considerati singolarmente. Javadoc non mostra le informazioni sulla versione, a meno che non sia specificata l'opzione `-version` su linea di comando.

Introdotto in JDK 1.0.

```
@param [nome del parametro] [descrizione]
```

Aggiunge il parametro specificato e la sua descrizione alla sezione `Parameters:` del metodo corrente.

Il commento doc riferito a un certo costruttore o a un certo metodo deve obbligatoriamente presentare un tag `@param` per ognuno dei parametri attesi, nell'ordine in cui l'implementazione del costruttore o del metodo specifica i parametri stessi. Può essere inserito solamente all'interno del codice sorgente Java dove non può essere usato per classi, interfacce o campi.

Introdotto in JDK 1.0.

```
@return [descrizione]
```

Aggiunge `Returns:` seguito dalla descrizione specificata.

Indica il tipo restituito e la gamma di valori possibili. Può essere inserito solamente all'interno del codice sorgente Java, dove deve essere obbligatoriamente usato per ogni metodo, a meno che questo non sia un costruttore o non presenti alcun valore di ritorno (`void`). Non può essere usato per classi, interfacce o campi.

Introdotto in JDK 1.0.

```
@exception [nome completo della classe] [descrizione]
```

Aggiunge `Throws:` seguito dal nome della classe specificata (che costituisce l'eccezione) e dalla sua descrizione.

Il commento doc riferito a un certo costruttore o a un certo metodo deve presentare un tag `@exception` per ognuna delle eccezioni che compaiono nella sua clausola `throws`, presentate in ordine alfabetico. Può essere inserito solamente all'interno del codice sorgente Java, dove non può essere usato per classi, interfacce o campi.

Introdotto in JDK 1.0.

```
@throws [nome completo della classe] [descrizione]
```

Stesso uso di `@exception`.
Introdotto in JDK 1.2.

```
@see [riferimento]
```

Aggiunge `See Also:` seguito dal riferimento indicato.
Può essere usato in qualunque commento doc, per tutti i possibili file sorgenti. Assume caratteristiche differenti, dipendenti dal carattere iniziale dell'argomento `[riferimento]`. In particolare, se si tratta di virgolette, ci si aspetta allora il nome di una qualunque risorsa stampata, e il riferimento è mostrato senza cambiamenti; se si tratta invece del segno di minore (<), si intende considerare il riferimento come un collegamento ipertestuale, inserendolo nella documentazione HTML con questo formato. Esiste infine una terza possibilità, in cui si rientra in tutti i restanti casi e per la quale il tag presenta la struttura

```
@see [riferimento] [nome]
```

In questo caso Javadoc mostra il nome indicato, rendendolo un collegamento ipertestuale al riferimento in questione (un package, una classe, un'interfaccia, un metodo, un costruttore o un campo). Se l'argomento `[nome]` risulta omesso (come capita di frequente), esso viene rimpiazzato dalla denominazione del riferimento. Il riferimento può assumere una delle seguenti forme:

- nome completo del package. Esempio: `@see java.lang`;
- nome completo della classe. Esempio: `@see java.lang.String`;
- nome della classe: Javadoc cerca la classe nel package corrente e tramite gli import. Esempio: `@see String`;
- nome (completo) della classe e nome del metodo: può crearsi ambiguità con metodi che effettuano l'overload e campi dello stesso nome. Se il nome della classe non risulta completo, Javadoc cerca la classe nel package corrente e tramite gli import. Esempio: `@see java.lang.String#charAt`;
- nome (completo) della classe e nome del metodo con parametri. Se il nome della classe non risulta completo, Javadoc cerca la classe nel package corrente e tramite gli import. Esempio: `@see java.lang.String#charAt(int)`;

- nome del metodo: per riferirsi a metodi della classe corrente. Esempio: `@see #charAt`;
- nome del metodo con parametri. Esempio: `@see #charAt(int)`;
- nome (completo) della classe e nome del campo. Se il nome della classe non risulta completo, Javadoc cerca la classe nel package corrente e tramite gli import. Esempio: `@see java.lang.String#CASE_INSENSITIVE_ORDER`;
- nome del campo: per riferirsi a campi della classe corrente. Esempio: `@see CASE_INSENSITIVE_ORDER`.

Introdotto in JDK 1.0.

`{@link [riferimento]}`

Stesso uso di `@see`.

L'unica differenza consiste nel non avere una sezione `See Also:`, ma bensì l'inserimento diretto del collegamento ipertestuale nello stesso punto in cui il tag `{@link}` viene indicato. Può essere usato in qualunque commento doc, comparendo sia nella sezione di descrizione che nella sezione dei tag, per tutti i possibili file sorgenti. Non esiste un limite al numero di tag `{@link}` utilizzabili.

Introdotto in JDK 1.2.

`@since [versione]`

Utilizzato per specificare da quale momento l'entità di riferimento (classe, interfaccia, metodo, costruttore, campo) è stata inserita nell'API.

Può essere usato in qualunque commento doc, per tutti i possibili file sorgenti. Questa informazione dovrebbe essere sempre fornita per tutte le classi e interfacce, e per tutti i metodi e i campi aggiunti dopo il primo rilascio della classe o interfaccia di appartenenza.

Introdotto in JDK 1.1.

`@serial [descrizione]`

Utilizzato per i campi serializzabili di default.

Può essere inserito solamente all'interno del codice sorgente Java. La descrizione non

è obbligatoria: se presente, deve spiegare il significato del campo ed elencare i valori accettabili.

Introdotto in JDK 1.2.

```
@serialField [nome] [tipo] [descrizione]
```

Può essere inserito solamente all'interno del codice sorgente Java. Ogni classe serializzabile può definire il proprio formato serializzato dichiarando, in un campo nominato `serialPersistentFields`, un array di oggetti di tipo `ObjectStreamField`. In questo caso, il commento doc di `serialPersistentFields` deve includere un tag `@serialField`, e i rispettivi argomenti (`[nome]`, `[tipo]`, `[descrizione]`), per ogni elemento dell'array.

Introdotto in JDK 1.2.

```
@serialData [descrizione]
```

Può essere inserito solamente all'interno del codice sorgente Java. Ogni classe serializzabile può definire un metodo `writeObject()` per trasmettere dati aggiuntivi rispetto a quelli trasferiti di default dal meccanismo di serializzazione. Ogni classe esternalizzabile può definire un metodo `writeExternal()`, responsabile della completa scrittura dello stato dell'oggetto sul canale di serializzazione. Il tag `@serialData` deve essere usato nei commenti doc per i metodi `writeObject()` e `writeExternal()`. La descrizione specifica le caratteristiche del meccanismo di serializzazione.

Introdotto in JDK 1.2.

```
@deprecated [spiegazione]
```

Aggiunge `Deprecated` seguito dalla spiegazione indicata.

Può essere inserito in qualunque commento doc, per tutti i possibili file sorgenti. Deve essere usato per classi, metodi, costruttori e campi che risultano deprecati (e il cui utilizzo è quindi sconsigliato). La spiegazione deve riportare, nella sua prima frase, il momento di attivazione del provvedimento e indicare un possibile sostituto. Il compilatore Javac annota i tag `@deprecated` per poter avvisare le classi che utilizzano elementi deprecati.

Introdotto in JDK 1.0.

Alcune fonti riportano la presenza di un tag ulteriore, indicato come `@beaninfo`, che non è considerato standard e non viene in realtà usato da Javadoc: il suo scopo, una volta

processato da un altro strumento di Sun, è quello di generare la classe `java.beans.BeanInfo`. Esiste un documento di Sun, *How to Write Doc Comments for Javadoc* (si veda in bibliografia), il quale fornisce tra le altre cose una serie di suggerimenti e convenzioni per una corretta scrittura della sezione dei tag.

JS: opzioni

Javadoc presenta un ampio elenco di opzioni da linea di comando, raggruppabili in due insiemi distinti:

- opzioni utilizzabili da tutti i doclet;
- opzioni utilizzabili esclusivamente dallo standard doclet.

Opzioni per tutti i doclet

```
-1.1
```

Genera la documentazione con l'aspetto e le funzionalità della documentazione prodotta da Javadoc 1.1 (si veda *What's New in Javadoc 1.2* in bibliografia). Non tutte le opzioni qui presentate funzionano con Javadoc 1.1: è consigliabile effettuare la verifica di compatibilità eseguendo da linea di comando

```
javadoc -1.1 -help
```

```
-bootclasspath [locazione delle classi (classpath)]
```

Specifica il percorso per raggiungere le classi di boot, quelle cioè appartenenti ai cosiddetti core package.

```
-classpath [locazione delle classi (classpath)]
```

Specifica il percorso per raggiungere le classi Java. Se l'opzione `-sourcepath` non è presente, Javadoc usa il classpath (indicato dall'opzione `-classpath` o dal sistema operativo) per la ricerca dei file sorgenti. In assenza del classpath, la ricerca delle classi avviene nella directory corrente.

```
-doclet [nome della classe]
```

Specifica la classe del doclet che si intende utilizzare al posto dello standard doclet.

`-docletpath [locazione delle classi (classpath)]`

Specifica il percorso per raggiungere la classe specificata dall'opzione `-doclet`. È un'opzione inutile se la classe in questione risulta comunque raggiungibile.

`-encoding [nome del file]`

Specifica il nome del file sorgente per la codifica. Se l'opzione `-encoding` non è presente, Javadoc usa il meccanismo di default.

`-extdirs [elenco delle directory]`

Specifica le directory in cui risiedono le classi che usano il meccanismo Java Extension.

`-help`

Mostra l'help di Javadoc.

`-J[opzione java]`

Passa l'argomento indicato direttamente alla Java Virtual Machine. Non c'è alcuno spazio tra l'opzione e l'argomento.

`-locale [nome da java.util.Locale]`

Specifica le opzioni sulla lingua che Javadoc deve utilizzare per la documentazione generata. L'argomento da specificare è uno dei campi della classe `java.util.Locale`.

`-overview [nome o percorso completo del file]`

Specifica il file di commento generale.

`-package`

Mostra solamente classi, metodi e campi con modificatore di visibilità `package`, `protected` e `public`.

`-private`

Mostra tutte le classi, i metodi e i campi.

`-protected`

Mostra solamente classi, metodi e campi con modificatore di visibilità `protected` e `public`. Questa è l'opzione di default.

`-public`

Mostra solamente classi, metodi e campi con modificatore di visibilità `public`.

`-sourcepath [locazione dei sorgenti]`

Specifica il percorso per raggiungere i file sorgenti Java quando, dopo il comando `javadoc`, è presente l'argomento `[package]` (si veda sezione "JS: utilizzo"). Non viene utilizzato per i file sorgenti Java elencati dall'argomento `[file sorgenti]` (si veda sezione "JS: utilizzo"). Se l'opzione `-sourcepath` non è presente, Javadoc usa il classpath (indicato dall'opzione `-classpath` o dal sistema operativo) per la ricerca dei file sorgenti. In assenza del classpath, la ricerca dei file sorgenti avviene nella directory corrente.

`-verbose`

Fornisce messaggi più dettagliati durante l'esecuzione di Javadoc, riguardanti in particolare il tempo impiegato per processare ogni file sorgente Java.

Opzioni per lo standard doclet

`-author`

Permette di visualizzare il tag `@author` nella documentazione generata.

`-bottom [piè di pagina]`

Specifica il piè di pagina da inserire in fondo ad ogni file generato, al di sotto della barra di navigazione. Il piè di pagina può contenere tag HTML e spazi bianchi, a patto di essere racchiuso tra virgolette.

`-charset [nome]`

Specifica il set di caratteri HTML adottato dalle pagine HTML generate.

`-d [nome della directory]`

Specifica, in modo assoluto oppure relativo, la directory di destinazione nella quale Javadoc salva i file HTML generati. Se l'opzione `-d` non è presente, i file generati vengono salvati nella directory corrente.

`-docencoding [nome]`

Specifica la codifica per le pagine HTML prodotte.

`-doctitle [titolo]`

Specifica il titolo da inserire in cima al file `overview-summary.html` (si veda sezione "JS: file generati"). Il titolo può contenere tag HTML e spazi bianchi, a patto di essere racchiuso tra virgolette.

`-footer [piè di pagina]`

Specifica il piè di pagina da inserire in fondo ad ogni file generato, sulla destra della barra di navigazione. Il piè di pagina può contenere tag HTML e spazi bianchi, a patto di essere racchiuso tra virgolette.

-group [titolo] [elenco dei package]

Suddivide l'elenco delle voci presenti nella pagina contenente l'intero insieme dei package documentati (overview-summary.html, si veda "JS: file generati"), creando una serie di gruppi di package logicamente correlati. L'argomento [titolo] indica il titolo di uno di questi gruppi, le cui voci (separate da punto e virgola) sono riportate nell'argomento [elenco dei package]. Non esiste un limite al numero di opzioni -group inseribili.

-header [intestazione]

Specifica l'intestazione da inserire in cima ad ogni file generato. L'intestazione può contenere tag HTML e spazi bianchi, a patto di essere racchiusa tra virgolette.

-helpfile [nome o percorso completo del file]

Specifica il percorso di un file alternativo di aiuto, al quale indirizzare il link HELP presente sulla barra di navigazione. Il nome di questo file può essere scelto liberamente. Se l'opzione -helpfile non è presente, Javadoc crea il file help-doc.html (si veda sezione "JS: file generati").

-link [url]

Specifica un URL verso un set di documentazione che, prodotto da Javadoc in un momento precedente l'esecuzione corrente, si vuole rendere raggiungibile dalle pagine che Javadoc si appresta a generare. L'URL può essere assoluto oppure relativo. La directory indicata dall'argomento [url] deve contenere il file package-list che Javadoc necessita di leggere a runtime (si veda sezione "JS: file generati"). Se l'URL si trova sul World Wide Web, Javadoc deve poter sfruttare una connessione web per accedere al file package-list. Se l'opzione -link non è presente, i riferimenti a classi e membri che risultano esterni non possiedono il relativo collegamento ipertestuale.

-linkoffline [url] [elenco dei package]

Ha le stesse caratteristiche dell'opzione -link, fatta eccezione per il fatto che l'elenco dei package è specificato su linea di comando e non nel file package-list. È utilizzato

quando la directory indicata dall'argomento `[url]` non contiene il file `package-list` o nel caso in cui questo file non sia raggiungibile da Javadoc durante la sua esecuzione.

`-nodeprecated`

Impedisce la creazione di documentazione per le entità deprecate (si veda sezione "JS: file generati").

`-nodeprecatedlist`

Impedisce la creazione del file `deprecated-list.html` (si veda sezione "JS: file generati") e del corrispondente link sulla barra di navigazione.

`-nohelp`

Impedisce la creazione del file di help (si veda sezione "JS: file generati") e del corrispondente link sulla barra di navigazione.

`-noindex`

Impedisce la creazione del file di indice (si veda sezione "JS: file generati").

`-nonavbar`

Impedisce la creazione della barra di navigazione, sia in cima che in fondo alle pagine generate.

`-notree`

Impedisce la creazione di documentazione per la gerarchia delle classi o delle interfacce (si veda sezione "JS: file generati").

`-serialwarn`

Genera, al momento della compilazione, eventuali warning dovuti a tag `@serial` mancanti.

`-splitindex`

Suddivide il file di indice (si veda sezione "JS: file generati") in molteplici file, uno per ogni diversa lettera iniziale incontrata nell'indice e uno per ogni voce che non inizia con una lettera dell'alfabeto.

`-stylesheetfile [nome o percorso completo del file]`

Specifica il percorso di uno stylesheet alternativo. Il nome di questo file può essere scelto liberamente. Se l'opzione `-stylesheetfile` non è presente, Javadoc crea il file `stylesheet.css` (si veda sezione "JS: file generati").

`-title [titolo]`

Opzione non più esistente: è stata rinominata in `-doctitle`.

`-use`

Genera una pagina di "utilizzo" per ogni package (`package-use.html`) e una per ogni classe o interfaccia (`class-use/MiaClasse.html`) documentate. Queste pagine descrivono quali package, classi, metodi, costruttori e campi utilizzano parti del package, della classe o dell'interfaccia in questione, in termini di classi da estendere, tipi di campi, tipi di valori di ritorno per i metodi, tipi di parametri per i metodi e i costruttori.

`-version`

Permette di visualizzare il tag `@version` nella documentazione generata.

`-windowtitle [titolo]`

Specifica il titolo da inserire nel tag HTML `<TITLE>`. Il titolo non può contenere alcun

tag HTML. Se l'opzione `-windowtitle` non è presente, Javadoc utilizza il valore dell'opzione `-doctitle`.

Bibliografia

DAVID FLANAGAN, *Java In A Nutshell*, O'Reilly, 3ª edizione

Javadoc Tool Home Page: `http://java.sun.com/j2se/javadoc/`

Javadoc FAQ: `http://java.sun.com/j2se/javadoc/faq.html`

How to Write Doc Comments for Javadoc: `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`

Javadoc – The Java API Documentation Generator: `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/javadoc.html`

Doclet Overview: `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html`

What's New in Javadoc 1.2: `http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/whatsnew.html`

What's New in Javadoc 1.3: `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/whatsnew-1.3.html`

Indice analitico